



UNIVERSIDAD
DE GRANADA

Informática Gráfica: Teoría. Tema 2. Modelos de objetos.

Carlos Ureña

2022-23

Grado en Informática y Matemáticas

Grado en Informática y Administración y Dirección de Empresas

Dpt. Lenguajes y Sistemas Informáticos

ETSI Informática y de Telecomunicación

Universidad de Granada

Teoría. Tema 2. Modelos de objetos.

Índice.

1. Modelos geométricos. Introducción.
2. Modelos de fronteras: mallas de polígonos.
3. Representación en memoria de modelos de fronteras.
4. Transformaciones geométricas
5. Modelos jerárquicos. Representación y visualización.

Sección 1.
Modelos geométricos. Introducción..

Modelos geométricos formales

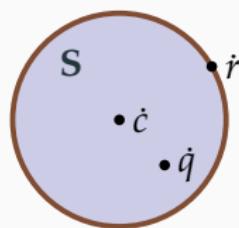
Un **modelo geométrico** es un modelo matemático abstracto que sirve para representar un objeto geométrico que existe en un espacio afín **E** (2D o 3D).

- ▶ Los modelos deben permitir la visualización computacional de los objetos que representan.
- ▶ Los más usados hoy en día son los **modelos de fronteras**: son estructuras de datos que representan la frontera del objeto de forma exacta o aproximada, normalmente mediante **mallas de triángulos**, pero hay otras posibilidades.
- ▶ Un modelo alternativo son los **modelos de volúmenes**.
- ▶ Ultimamente se usan bastante modelos basados en las **funciones de distancia con signo** (*signed distance functions*) o SDFs. Cada objeto se representa con un algoritmo que calcula la distancia (o una cota) desde cualquier punto al objeto.

En la asignatura nos centramos en las mallas de triángulos.

Los conjuntos de puntos

Los modelos geométricos matemáticos abstractos más generales posibles son los **subconjuntos de puntos** de \mathbf{E} , por ejemplo, una esfera:



$$\begin{aligned}S &= \{ p \in \mathbf{E} \text{ t.q. } \|p - c\| \leq 1 \} \\p &\in S \\r &\in S \quad r \in \partial S \\\dot{s} &\notin S\end{aligned}$$

Cada subconjunto o **región S** es **cerrado** (incluye a su propia **superficie o frontera, ∂S**), además:

- ▶ Es **acotado** (no tiene extensión infinita),
- ▶ Su superficie es diferenciable (**plana** a escala muy pequeña)

Representaciones computacionales

El modelo basado en subconjuntos de puntos del espacio:

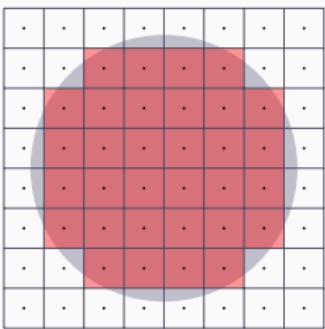
- ▶ permite representar matemáticamente cualquier objeto (es el modelo **más general** posible)
- ▶ pero **no se puede representar en la memoria** (finita, discreta) de un ordenador

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

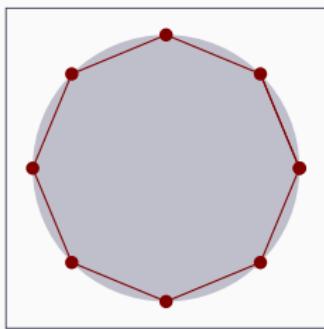
- ▶ **Enumeración espacial:** se partitiona el espacio en celdas (llamados **voxels**), y cada una se clasifica como interior o exterior al objeto.
- ▶ **Modelos de fronteras:** se representa la frontera (la superficie) en lugar de todo el interior, para ello se usan conjuntos finitos de polígonos planos adyacentes entre ellos (**caras**)

Ejemplo 2D de los modelos aproximados

Las dos formas computacionales de representar objetos son aproximadamente iguales al modelo ideal basado (un subconjunto), con un error que disminuye al aumentar la cantidad de memoria usada (la precisión o resolución).



Enumeración espacial

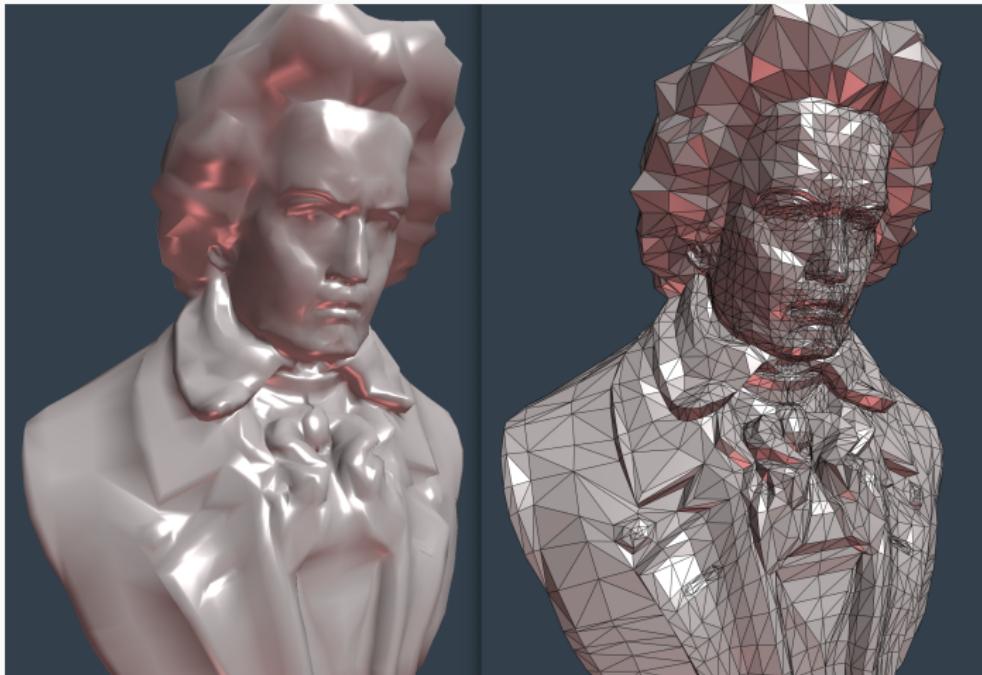


Modelos de fronteras

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

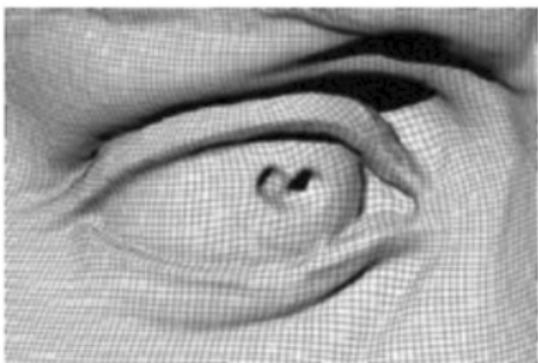
Ejemplos de modelos de fronteras 3D (1/2)

Ejemplo de una **mallas de polígonos** (de las prácticas). A la izquierda el modelo con iluminación, a la derecha vemos las caras que forman el modelo:



Ejemplos de modelos de fronteras 3D (2/2)

Los modelos de fronteras (a muy alta resolución) permiten representar fielmente casi cualquier objeto real:



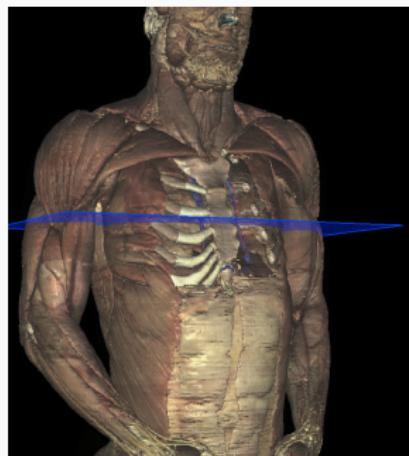
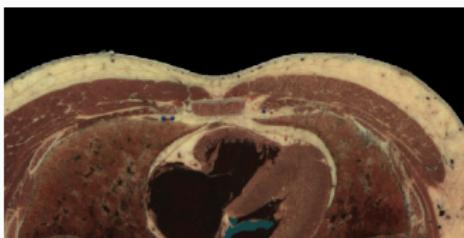
Escaneo 3D del *David* de Miguel Angel en Florencia en 1999.

Marc Levoy et al. The Digital Michelangelo Project:

☞ <https://accademia.stanford.edu/mich/>

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. Elementos y adyacencia.
- 2.2. Atributos de vértices.

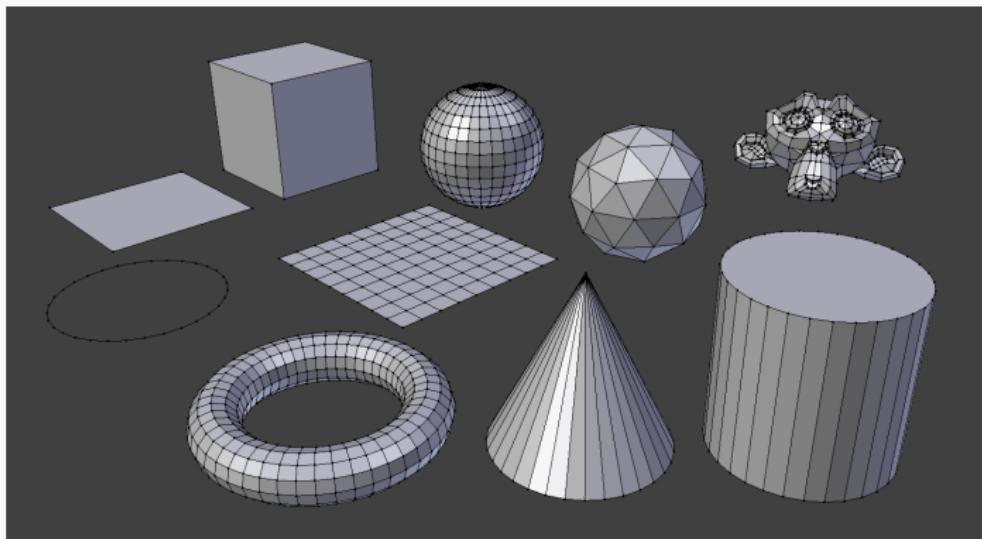
Mallas de polígonos.

Una **malla de polígonos** (*polygon Mesh*) es un conjunto de puntos de un espacio afín que forman **caras** (*faces*) planas, usualmente adyacentes entre ellas, y que aproxima la frontera de un objeto en el espacio 3D

- ▶ El término *objeto* designa un conjunto de puntos como los descritos antes (de extensión finita, continuo). Esos puntos pertenecen todos a un mismo espacio afín.
 - ▶ Una cara es un conjunto de puntos en un plano de dicho espacio afín, delimitados por un polígono.
 - ▶ Las mallas aproxima una superficie, la cual
 - ▶ encierra completamente una región del espacio (el objeto tiene volumen), o bien
 - ▶ constituye en si misma el objeto, que tiene volumen nulo.
- Las primeras son mallas *cerradas* y las segundas *abiertas*.

Ejemplos de mallas

Aquí vemos varios ejemplos de mallas de polígonos (excepto la circunferencia que no lo es). Algunas son cerradas y otras abiertas:

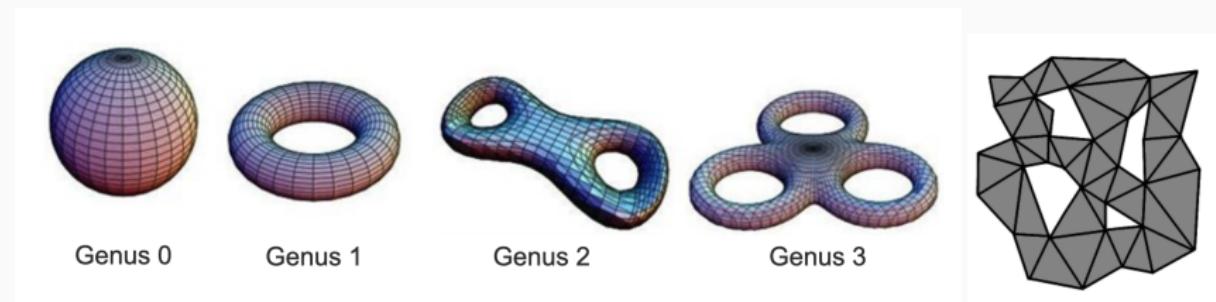


Catálogo de objetos predefinidos de la aplicación *Blender* para modelado 3D:

☞ <https://docs.blender.org/manual/en/latest/modeling/meshes/primitives.html>

Características de las mallas

- ▶ Las mallas cerradas pueden tener cualquier *género topológico* (*genus*), aquí vemos mallas de género 0, 1, 2 y 3.
- ▶ Las mallas abiertas pueden tener huecos entre los polígonos:



Izquierda: Rudiger Westermann (Univ. Munich) - Computer Graphics Course slides

↗ <https://slideplayer.com/slide/4642205/>

Derecha: ↗ Stackoverflow

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

Sección 2. Modelos de fronteras: mallas de polígonos.

Subsección 2.1.

Elementos y adyacencia..

Elementos de las mallas: vértices

Un **vértice** (*vertex*) es un par formado por un **punto** del espacio afín (en el extremo de alguna una arista), y un **valor entero único** (entre 0 y $n - 1$, donde n es el número de vértices de la malla).

- ▶ Al punto lo llamamos la **posición** del vértice.
- ▶ Al entero lo llamamos **índice** del vértice.
- ▶ Dos vértices distintos (con distinto índice) pueden tener la misma posición.
- ▶ Usar estos índices:
 - ▶ permiten expresar la *topología* de una malla independientemente de su *geometría*.
 - ▶ facilita construir representaciones computacionales de las mallas con ciertas propiedades.

Elementos de las mallas: caras

Una **cara** (*face*) contiene un conjunto de puntos coplanares que están delimitados por un único polígono plano. Se determina por una secuencia ordenada de índices de vértices que forman dicho polígono.

- ▶ En la secuencia de índices, cada vértice comparte una arista con el siguiente (y el último con el primero). En esta secuencia
 - ▶ es indiferente cual índice es el primero.
 - ▶ en principio, es indiferente en que sentido se recorren los vértices (solo hay dos posibilidades).
- ▶ Dos caras distintas no pueden tener asociado el mismo conjunto de índices, ni siquiera con distinto orden o empezando en distintos vértices.

Elementos de las mallas: aristas. Representación de mallas.

Una **arista** (*edge*) contiene el conjunto de puntos en un lado del polígono que delimita una cara, puntos que forman un segmento de recta. Se determina por un par único de índices de vértices.

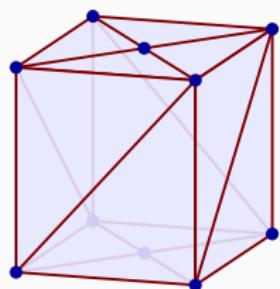
- ▶ Los dos índices de vértice de una arista no pueden coincidir.
- ▶ El orden en el que aparecen los índices en el par es irrelevante (las aristas no están orientadas).
- ▶ Dos aristas distintas no pueden tener el mismo par de índices de vértice, ni siquiera en distinto orden.

Esto implica que una malla viene determinada por:

- ▶ la secuencia $\{\dot{p}_0, \dot{p}_1, \dots, \dot{p}_{n-1}\}$ de posiciones de sus n vértices.
- ▶ la secuencia de caras, cada una de ellas representada como una secuencia de k índices de vértice: $\{i_0, \dots, i_{k-1}\}$ (k puede ser distinto en cada cara).

Vértices, caras y aristas

Elementos de una malla que forma la frontera de un paralelepípedo:



vertices



edges



faces



[Wikipedia: Polygon Mesh.](#)

Adyacencia entre elementos de una malla

En una malla existen relaciones binarias de adyacencia entre estos elementos:

- ▶ Un vértice en el extremo de una arista es adyacente a la arista (por tanto, toda arista es adyacente a exactamente dos vértices).
- ▶ Una arista y una cara son adyacentes si la arista forma parte del polígono que delimita la cara.
- ▶ Dos vértices son adyacentes si hay una arista adyacente a ambos.
- ▶ Dos caras son adyacentes si hay una arista adyacente a ambas.
- ▶ Un vértice y una cara son adyacentes si hay una arista adyacente a ambos.

Puesto que los vértices están numerados, las relaciones de adyacencia se pueden expresar en términos de los índices de los vértices.

Geometría y topología de las mallas

Una malla tiene una geometría y una topología:

- ▶ **Geometría:** conjunto de puntos que están en alguna cara (eso incluye los puntos que están en alguna arista y las posiciones de los vértices).
- ▶ **Topología:** conjunto de relaciones de adyacencia entre vértices, aristas y caras (sin tener en cuenta la geometría, es decir, considerando únicamente los índices de los vértices).

Esta definición permite que dos mallas distintas

- ▶ tengan la misma topología pero distinta geometría (p.ej., partimos de una malla y cambiamos las posiciones de sus vértices, pero mantenemos las adyacencias).
- ▶ tengan la misma geometría pero distinta topología (p.ej., partimos de una malla con caras de cuatro aristas y dividimos cada cara en dos caras triángulares coplanares).

Características de las 2-variedades

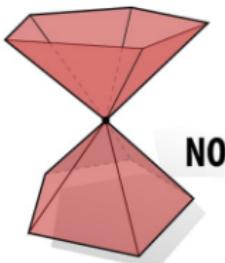
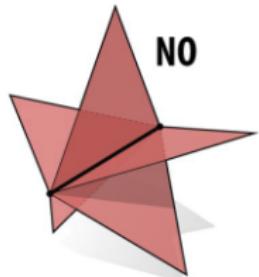
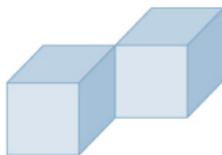
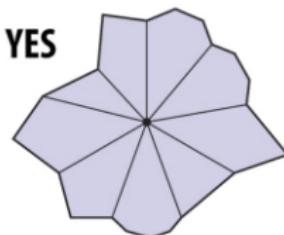
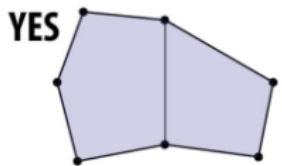
Vamos a usar exclusivamente mallas que son una **2-variedad** (**2-manifold**), esto implica que:

- ▶ Un vértice siempre es adyacente a dos aristas como mínimo (no hay vértices aislados).
- ▶ Una arista siempre es adyacente a una o a dos caras (no hay aristas aisladas, ni aristas adyacentes a 3 o más caras).
- ▶ Todas las caras adyacentes a un vértice se pueden ordenar en una secuencia en la cual cada cara es adyacente a la siguiente.

Estas propiedades aseguran, entre otras cosas, que se pueden asignar ciertos atributos a cada vértice de forma única (por ejemplo, normales y coordenadas de textura), ya que el entorno de un punto de la superficie siempre es *equivalente* a un plano.

Ejemplos de mallas que no son 2-variedades

Solo son 2-variedades las mallas etiquetadas con yes:



Izquierda: Carnegie Mellon Computer Graphics Course: slides (fall 2018):

☞ <http://15462.courses.cs.cmu.edu/spring2018/lecture/meshes>

Derecha: J.F. Hughes et al.: Computer Graphics: Principles and Practice (3rd ed.)

Conversión en 2-variedad

La topología de una malla que no es una 2-variedad puede modificarse para que lo sea, manteniendo la geometría:

- ▶ Se puede conseguir replicando vértices, es decir, añadiendo nuevos vértices con índices distintos en la misma posición de otros vértices ya existentes (p.ej.: dos conos unidos por el ápice se separan al insertar dos ápices en la misma posición, un ápice por cono)
- ▶ Esto puede implicar a veces también replicar aristas (p.ej.: dos cubos unidos con una arista en común se separan duplicando los dos vértices de dicha arista y la propia arista).

Aristas y vértices de frontera

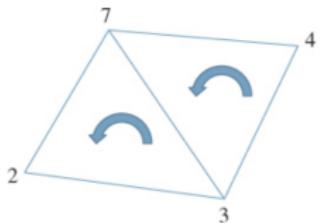
- ▶ Una arista es una **arista de frontera** (o de **borde**) (*boundary edge* o *border edge*) si es adyacente a una única cara.
- ▶ Un vértice es un vértice de frontera si es adyacente a alguna arista de frontera.
- ▶ Una malla es cerrada si y solo si no tiene aristas de frontera (todas las aristas son adyacentes a exactamente dos caras).
- ▶ Las mallas abiertas tienen al menos una cara de frontera.

Orientación coherente de vértices en cada cara

El orden de enumeración de los vértices en las caras debe ser coherente:

- ▶ Dos caras adyacentes tienen los vértices siempre enumerados en el mismo sentido (horario o antihorario).
- ▶ Por convenio, en una malla cerrada, una cara vista desde el exterior presenta sus vértices en sentido anti-horario.
- ▶ En las mallas abiertas, esto define dos *lados* de la superficie (desde uno se ven las caras en sentido anti-horario y desde el otro en sentido horario).

A modo de ejemplo, para enumerar los vértices de las dos caras



- ▶ es correcto: $(2, 3, 7)$ y $(4, 7, 3)$.
- ▶ es correcto: $(3, 2, 7)$ y $(7, 4, 3)$.
- ▶ es incorrecto: $(2, 3, 7)$ y $(4, 3, 7)$.

Conversión en 2-variedad

La topología de una malla que no es una 2-variedad puede modificarse para que lo sea, manteniendo la geometría:

- ▶ Se puede conseguir replicando vértices, es decir, añadiendo nuevos vértices con índices distintos en la misma posición de otros vértices ya existentes (p.ej.: dos conos unidos por el ápice se separan al insertar dos ápices en la misma posición, un ápice por cono)
- ▶ Esto puede implicar a veces también replicar aristas (p.ej.: dos cubos unidos con una arista en común se separan duplicando los dos vértices de dicha arista y la propia arista).

Marco de referencia de la malla.

La posición de cada vértice se representa en el ordenador 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 (en el punto \vec{p}_i) tiene coordenadas $\mathbf{c}_i = (x_i, y_i, z_i, 1)$ en el marco \mathcal{R} , es decir:

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

- ▶ A la tupla \mathbf{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).

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

Sección 2. Modelos de fronteras: mallas de polígonos.

Subsección 2.2.

Atributos de vértices..

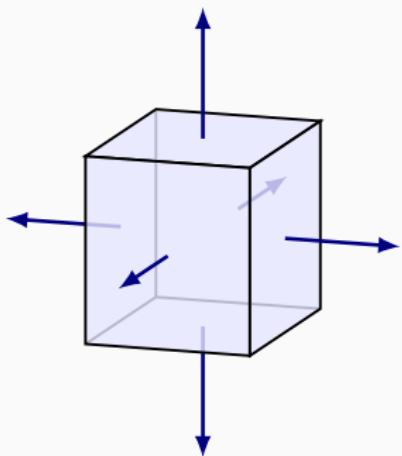
Atributos de las mallas

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: coords. de textura, vectores tangente y bitangente, etc...

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), o bien cuando se quiere hacer *sombreado plano*:



Para un polígono (con dos aristas \vec{a}, \vec{b} , vectores distintos, no nulos), su normal \vec{n} se define como:

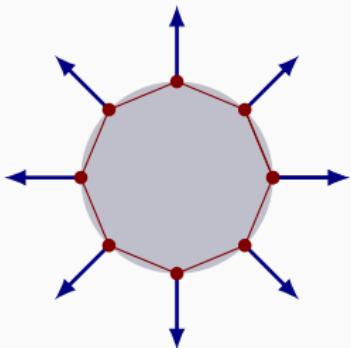
$$\vec{n} = \frac{\vec{m}}{\|\vec{m}\|} \quad \text{donde } \vec{m} = \vec{a} \times \vec{b}$$

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

Normales de vértices

Tienen sentido cuando la malla aproxima una superficie curvada:

- ▶ A veces la superficie original es conocida, y las normales se definen fácilmente (p.ej. una esfera).
- ▶ Si la superficie original es desconocida, las normales se pueden definir exclusivamente usando la malla



Para un vértice (con k caras adyacentes) su normal \vec{n} se define como:

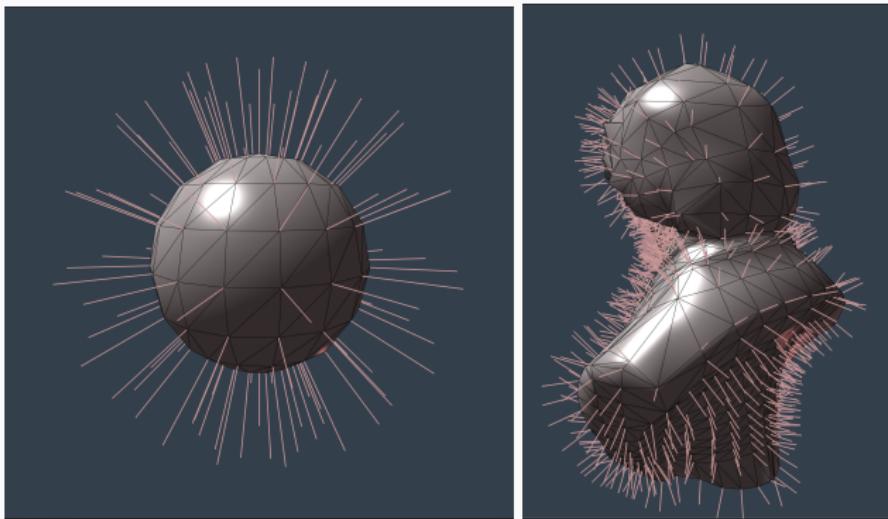
$$\vec{n} = \frac{\vec{s}}{\|\vec{s}\|} \text{ donde } \vec{s} = \sum_{i=0}^{k-1} \vec{m}_i$$

donde $\vec{m}_0, \vec{m}_1, \dots, \vec{m}_{k-1}$ son las normales de las caras adyacentes al vértice.

Las coordenadas de estas normales también se pueden precalcular y almacenar.

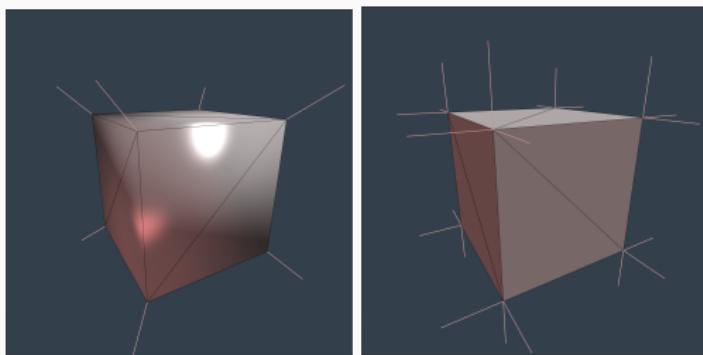
Ejemplos de normales de vértices calculadas

Aquí vemos visualizadas las normales de una esfera de baja resolución (calculadas analíticamente), y de una malla arbitraria (calculadas promediando normales de caras):



Discontinuidades de la normal

Algunos objetos reales presentan aristas o vértices donde la normal es discontinua (p.ej. un cubo), en ese caso promediar normales es mala idea, y es necesario replicar vértices y aristas (y después promediar):



El cubo de la izquierda tiene 8 vértices y el de la derecha 24 vértices. La iluminación es correcta a la derecha. El mismo problema puede aparecer con las coordenadas de textura, o los colores.

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:

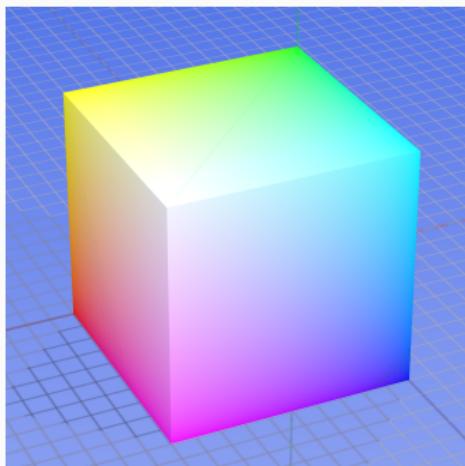


Imagen:  http://en.wikipedia.org/wiki/File:RGB_color_solid_cube.png
(Wikimedia Commons)

Sección 3. Representación en memoria de modelos de fronteras..

- 3.1. Triángulos aislados (TA).
- 3.2. Tiras de triángulos (TT).
- 3.3. Mallas indexadas.
- 3.4. Representación con estructura de aristas aladas

Representación en memoria

En esta sección veremos distintas formas de representar las mallas en la memoria de un ordenador:

- ▶ **Triángulos aislados, tiras de triángulos:** no representan explícitamente la topología.
- ▶ **Mallas indexadas:** lo más común, representan explícitamente la topología.
- ▶ **Aristas aladas:** extensión de las mallas indexadas para eficiencia en tiempo.

OpenGL está diseñado para visualizar directamente los triángulos aislados, las tiras de triángulos y las mallas indexadas.

Por simplicidad, nos restringimos a **caras triangulares**, que es lo más común en la inmensa mayoría de las aplicaciones.

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

Sección 3. Representación en memoria de modelos de fronteras.

Subsección 3.1.

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 tres entradas (tres variables de tipo **Tupla3f**) para cada triángulo:

Malla TA (n triángulos)		
0	x_0	y_0
1	x_1	y_1
2	x_2	y_2
3	x_3	y_3
4	x_4	y_4
5	x_5	y_5
:	:	
$3n - 3$	x_{3n-3}	y_{3n-3}
$3n - 2$	x_{3n-2}	y_{3n-2}
$3n - 1$	x_{3n-1}	y_{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.

Triángulos aislados: valoración.

Esta representación es poco eficiente en tiempo y memoria:

- ▶ Si un vértice es adyacente a k triángulos, sus coordenadas aparecen repetidas k veces en la tabla y se procesan k veces al visualizar.
- ▶ 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.

Además, no hay información explícita sobre la **topología** de la malla:

- ▶ La topología se puede calcular comparando coordenadas de vértices, pero tendría una complejidad en tiempo cuadrática con el número de vértices y es poco robusto.

En algunos casos muy particulares, podría ser útil (objetos realmente compuestos de muchos triángulos realmente aislados, objetos muy sencillos).

Implementación de mallas: clase base abstracta.

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

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

- ▶ 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.

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

Implementación de mallas de triángulos aislados

Como ejemplo, podemos usar una clase (**MallaTA**), con un vector con las coordenadas (contiguas) en memoria. En total, para n_t triángulos, se guardan $9n_t$ valores reales:

```
class MallaTA : public Objeto3D
{ protected:
    std::vector<Tuple3f> vertices ;           // tabla de vértices (3nt tuplas)
    GLenum nombre_vao = 0 ; // VAO (creado en 1a visualización)
    // .... (otros métodos)
public:
    virtual void visualizarGL( ContextoVis & cv ) ;
} ;
```

La visualización puede hacerse considerando la secuencia de vértices como una **secuencia no indexada** (no hay índices) con **glDrawArrays**, usando **GL_TRIANGLES** como tipo de primitiva. En la instancia se guarda el nombre del VAO.

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

Sección 3. Representación en memoria de modelos de fronteras.

Subsección 3.2.

Tiras de triángulos (TT)..

Tiras de triángulos: motivación y características

Las representación en memoria usando **tiras de triángulos** pretende reducir la memoria y el tiempo que necesitan la representación de triángulos aislados:

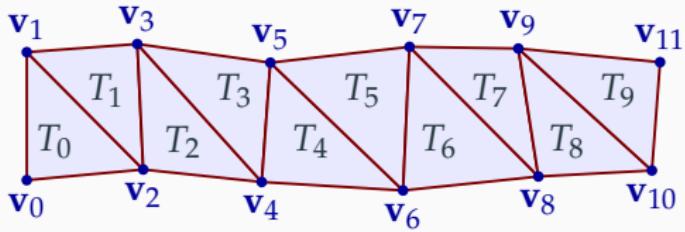
- ▶ Para conseguir esto, esta representación reduce el número de veces que aparecen replicadas unas coordenadas en memoria.
- ▶ Como consecuencia, se reduce el tiempo de procesamiento.

Sin embargo, esta representación

- ▶ No evita totalmente las redundancias.
- ▶ Tampoco incluye información explícita sobre la topología de la malla.

Tiras de triángulos en una malla.

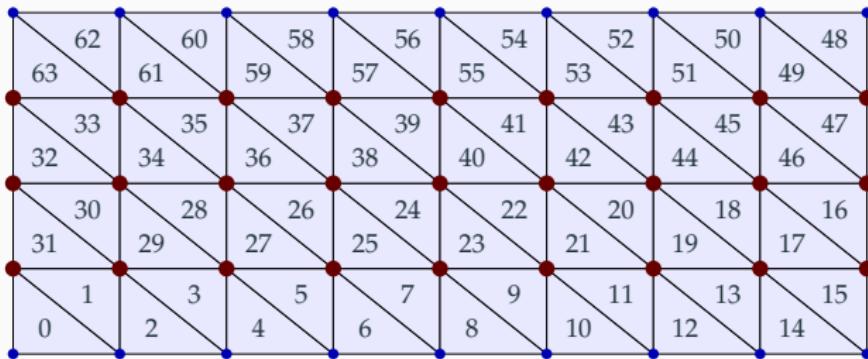
Podemos identificar una parte de una malla como una **tira de triángulos**: cada triángulo T_{i+1} en la secuencia es adyacente al anterior T_i , con lo cual T_{i+1} comparte con T_i una arista y dos vértices, vértices cuyas coordenadas no tienen que ser repetidas en memoria:



- ▶ 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

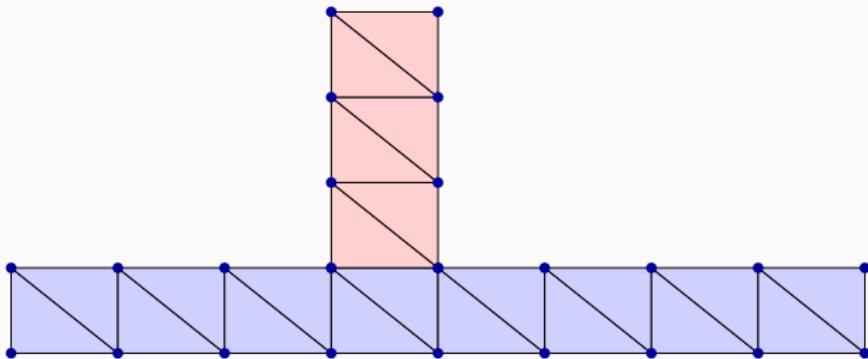
En la mayoría de los casos, las tiras obligan a repetir algunas coordenadas de vértices (aunque en mucho menor grado que los triángulos aislados). Ejemplo de una tira en zig-zag:



- ▶ 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 prever 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
:	:	:
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
:	:	:
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
:	:	:
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.

Tiras de triángulos: Implementación

Una malla de tiras de triángulos se puede representar con una instancia de **MallaTT**, que a su vez tiene una o varias tiras (en **TiraTri**):

```
struct TiraTri // Tira de triángulos (coords. de vértices)
{
    unsigned long      num_tri; // número de triángulos en esta tira
    std::vector<Tupla3f> ver ; // coords. de vert. (num_tri+2 entradas)
    GLuint             nombre_vao = 0; // VAO (creado en 1a visu.)
} ;
class MallaTT : public Objeto3D // Malla compuesta de tiras de triángulos
{
protected:
    std::vector<TiraTri> tiras; // vector de tiras
    ....
public:
    virtual void visualizarGL( ContextoVis & cv );
} ;
```

Cada tira puede tener su propio VAO y se visualiza con **glDrawArrays** usando **GL_TRIANGLE_STRIP** como tipo de primitiva.

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

Sección 3. Representación en memoria de modelos de fronteras.

Subsección 3.3.

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:

Tabla Triángulos (n tri.)

$i_{0,0}$	$i_{0,1}$	$i_{0,2}$
$i_{1,0}$	$i_{1,1}$	$i_{1,2}$
$i_{2,0}$	$i_{2,1}$	$i_{2,2}$
$i_{3,0}$	$i_{3,1}$	$i_{3,2}$
\vdots	\vdots	\vdots
$i_{n-2,0}$	$i_{n-2,1}$	$i_{n-2,2}$
$i_{n-1,0}$	$i_{n-1,1}$	$i_{n-1,2}$

$$0 \leq i_{jk} < m$$

$$i_{1,2} = 3$$

Tabla Vértices (m verts.)

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
\vdots	\vdots	\vdots	\vdots
\vdots	\vdots	\vdots	\vdots
$m-2$	x_{m-2}	y_{m-2}	z_{m-2}
$m-1$	x_{m-1}	y_{m-1}	z_{m-1}

Implementación de las mallas indexadas

Usaremos una clase de nombre **MallaInd**:

```
class MallaInd : public Objeto3D
{
protected:
    std::vector<Tuple3f> vertices ; // tabla de vértices
    std::vector<Tuple3u> triangulos; // tabla de triángulos (índices)
    GLuint             nombre_vao = 0; // VAO
    .....
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.

Se visualiza con **glDrawElements** usando un VAO y **GL_TRIANGLES** como tipo de primitiva.

Tiras de triángulos indexadas

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.

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 es 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
```

Archivos con mallas indexadas: el formato OBJ

El formato OBJ se usa bastante hoy en día, es parecido a PLY, pero **con las normales y coordenadas de textura indexadas**:

- ▶ Incluye una tabla de vértices y una tabla de triángulos, igual que PLY
- ▶ Además, incluye tablas normales y coordenadas de textura. A diferencia de PLY, su tamaño no tiene porque coincidir con el de la tabla de vértices.
- ▶ En cada cara, cada vértice se representa por un índice de sus coordenadas de posición, y opcionalmente, otros índices independientes para su normal y sus coordenadas de textura.

Formato OBJ: valoración

La ventaja frente a PLY (malla indexada de triángulos) es una mayor flexibilidad lo que permite mayor eficiencia en memoria:

- ▶ dos vértices en posiciones distintas pueden compartir normal (por ejemplo, una malla plana puede tener una única normal, en lugar de tantas como vértices)
- ▶ un vértice único puede tener distintas coords. de textura o distintas normales en distintas caras, no es necesario replicarlo (por ejemplo, un cubo puede tener 8 vértices y solo 6 normales).

La principal desventaja es que OpenGL no puede visualizar directamente este tipo de tablas, así que es necesario convertirlas a una malla indexada de triángulos, replicando normales y coordenadas de textura.

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.

Problema: comparación de eficiencia en memoria (1/2)

Problema 2.1.

Supongamos que queremos codificar una esfera de radio $1/2$ y centro en el origen de dos formas:

- ▶ Por enumeración espacial, dividiendo el cubo que engloba a la esfera en celdas, de forma que haya k celdas por lado del cubo, todas ellas son cubos de $1/k$ de ancho. Cada celda ocupa un bit de memoria (si su centro está en la esfera, se guarda un 1, en otro caso un 0).
- ▶ Usando un modelo de fronteras (una malla indexada de triángulos), en el cual se usa una rejilla de triángulos y aristas que siguen los meridianos y paralelos, habiendo en cada meridiano y en cada paralelo un total de k vértices (se guarda únicamente la tabla de vértices y la de triángulos).

(continua en la siguiente transparencia)

Problema: comparación de eficiencia en memoria (2/2)

Problema 2.1. (continuación)

Asumiendo que un `float` y un `int` ocupan 4 bytes cada uno, contesta a estas cuestiones:

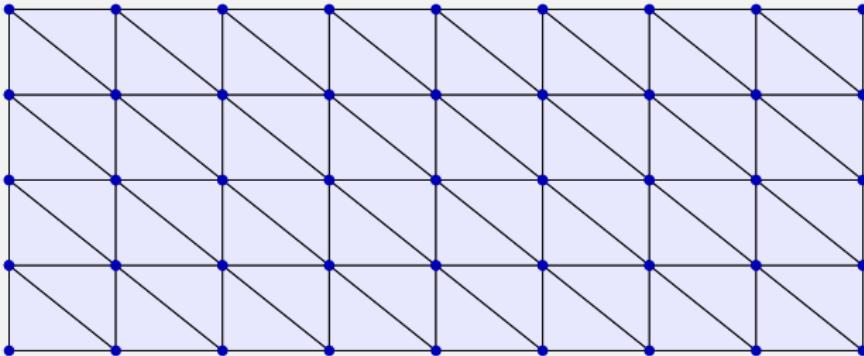
- ▶ Expresa el tamaño de ambas representaciones en bytes como una función de k .
- ▶ Suponiendo que $k = 16$ calcula cuantos KB de memoria ocupa cada estructura.
- ▶ Haz lo mismo asumiendo ahora que $k = 1024$ (expresa los resultados en MB)

Compara los tamaños de ambas representaciones en ambos casos ($k = 16$ y $k = 1024$).

Problema: uso de memoria en mallas indexadas (1/2)

Problema 2.2.

Considera una malla indexada (tabla de vértices y caras, esta última con índices de vértices) con topología de rejilla como la de la figura, en la cual hay n columnas de pares de triángulos y m filas (es decir, hay $n + 1$ filas de vértices y $m + 1$ columnas de vértices, con $n, m > 0$, en el ejemplo concreto de la figura, $n = 8$ y $m = 4$).



(continua en la siguiente transparencia)

Problema: uso de memoria en mallas indexadas (2/2)

Problema 2.2. (continuación)

En relación a este tipo de mallas, responde a estas dos cuestiones:

- (a) Supongamos que un **float** ocupa 4 bytes (igual a un **int**) ¿ que tamaño en memoria ocupa la malla completa, en bytes ? (tener en cuenta únicamente el tamaño de la tabla de vértices y triángulos, suponiendo que se almacenan usando los tipos **float** e **int**, respectivamente). Expresa el tamaño como una función de m y n .
- (b) Escribe el tamaño en KB suponiendo que $m = n = 128$.
- (c) Supongamos que m y n son ambos grandes (es decir, asumimos que $1/n$ y $1/m$ son prácticamente 0). deduce que relación hay entre el número de caras n_C y el número de vértices n_V en este tipo de mallas.

Problema: uso de memoria en tiras y mallas indexadas (1/2)

Problema 2.3.

Imagina de nuevo una malla como la del problema anterior, supongamos que usamos una representación como tiras de triángulos, de forma que cada fila de triángulos (con $2n$ triángulos) se almacena en una tira, habiendo un total de m tiras.

La tabla de punteros a tiras tiene un entero (el número de tiras) y m punteros, cada puntero suponemos que tiene 8 bytes de tamaño. De nuevo, asume que las coordenadas son de tipo **float** (4 bytes).

Responde a estas cuestiones:

(continua en la siguiente transparencia)

Problema: uso de memoria en tiras y mallas indexadas (2/2)

Problema 2.3. (continuación)

- (a) Indica que cantidad de memoria ocupa esta representación, en estos dos casos:
 - (a.1) Como función de n y m , en bytes.
 - (a.2) Suponiendo $m = n = 128$, en KB.
- (b) Para m y n grandes (es decir, cuando $1/n$ y $1/m$ son casi nulos), describe que relación hay entre el tamaño en memoria de la malla indexada del problema anterior y el tamaño de la malla almacenada como tiras de triángulos.
- (c) Si suponemos que la transformación de cada vértice se hace en un tiempo constante igual a la unidad, describe que relación hay entre los tiempos de procesamiento de vértices para esta malla cuando se representa como una malla indexada y como tiras de triángulos.

Problema: número de vértices, aristas y caras

Problema 2.4.

Supongamos una malla cerrada, simplemente conexa (topológicamente equivalente a una esfera), cuyas caras son triángulos y cuyas aristas son todas adyacentes a exactamente dos caras (la malla es un *poliedro* simplemente conexo de caras triangulares). Considera el número de vértices n_V , el número de aristas n_A y el número de caras n_C en este tipo de mallas.

Demuestra que cualquiera de esos números determina a los otros dos, en concreto, demuestra que se cumplen estas dos igualdades:

$$n_A = 3(n_V - 2)$$

$$n_C = 2(n_V - 2)$$

(nótese que, al igual que en el problema anterior, sigue siendo cierto que el número de caras es aproximadamente el doble que el de vértices).

Problema: creación de la tabla de aristas

Problema 2.5.

En una malla indexada, queremos añadir a la estructura de datos una tabla de aristas. Será un vector `ari`, que en cada entrada tendrá una tupla de tipo `Tupla2i` (contiene dos `int`) con los índices en la tabla de vértices de los dos vértices en los extremos de la arista. El orden en el que aparecen los vértices en una arista es indiferente, pero cada arista debe aparecer una sola vez.

Escribe el código de una función C++ para crear y calcular la tabla de aristas a partir de la tabla de triángulos. Intenta encontrar una solución con la mínima complejidad en tiempo y memoria posible. Suponer que el número de vértices adyacentes a uno cualquiera de ellos es como mucho un valor constante $k > 0$, valor que no depende del número total de vértices, que llamamos n .

(continua en la transparencia siguiente)

Problema: creación de la tabla de aristas

Problema 2.5. (continuación)

Considerar dos casos:

1. Los triángulos se dan con orientación *no coherente*: esto quiere decir que si un triángulo está formado por los vértices i, j, k , estos tres índices pueden aparecer en cualquier orden en la correspondiente entrada de la tabla de triángulos (puede aparecer como i, j, k o como i, k, j , o como k, j, i , etc....)
2. Los triángulos se dan con orientación *coherente*: esto quiere decir que si dos triángulos comparten una arista entre los vértices i y j , entonces en uno de los triángulos la arista aparece como (i, j) y en el otro aparece como (j, i) (decimos que en el triángulo a, b, c aparecen las tres aristas (a, b) , (b, c) y (c, a)). Además, asumimos que la malla es *cerrada*, es decir, que cada arista es compartida por exactamente dos triángulos.

Problema: cálculo del área de una malla indexada

Problema 2.6.

Escribe el pseudo-código de la función para calcular el área total de una malla indexada de triángulos, a partir de la tabla de vértices y de triángulos. Será una función que acepta un puntero a una **MallaInd** y devuelve un número real (asumir que se dispone del tipo **Tupla3f** y de los operadores usuales de tuplas o vectores, es decir suma **+**, resta **-**, producto escalar **·**, producto vectorial **×**, módulo **||** **||**, etc ...).

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

Sección 3. Representación en memoria de modelos de fronteras.

Subsección 3.4.

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 adyacencia 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 las mallas indexadas), mas una tabla de aristas (**tari**). Esta última:

- ▶ Tiene una entrada por cada arista, con dos índices:
 - ▶ **vi** = índice de vértice inicial
 - ▶ **vf** = índice de vértice final(es indiferente cual se selecciona como inicial y cual como final).
- ▶ Tambien tiene (cada arista es adyacente a dos triángulos):
 - ▶ **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 en O(1) sobre adyacencia arista-vértice y arista-triángulo.

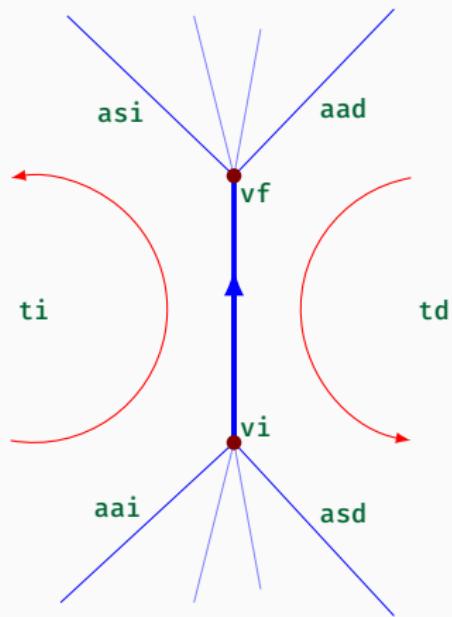
Aristas siguiente y anterior

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 hacer todas las consultas en $O(1)$, añadimos **taver** y **tatri**:

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

Sección 4. Transformaciones geométricas.

- 4.1. Concepto de transformación geométrica (TG).
- 4.2. Transformaciones usuales en IG.
- 4.3. Representación y operaciones sobre matrices.
- 4.4. Transformaciones en OpenGL con el cauce programable
- 4.5. Implementación de *modelview* en la clase **Cauce**.

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

Sección 4. Transformaciones geométricas

Subsección 4.1.

Concepto de transformación geométrica (TG)..

Coordenadas del mundo e instanciación de objetos.

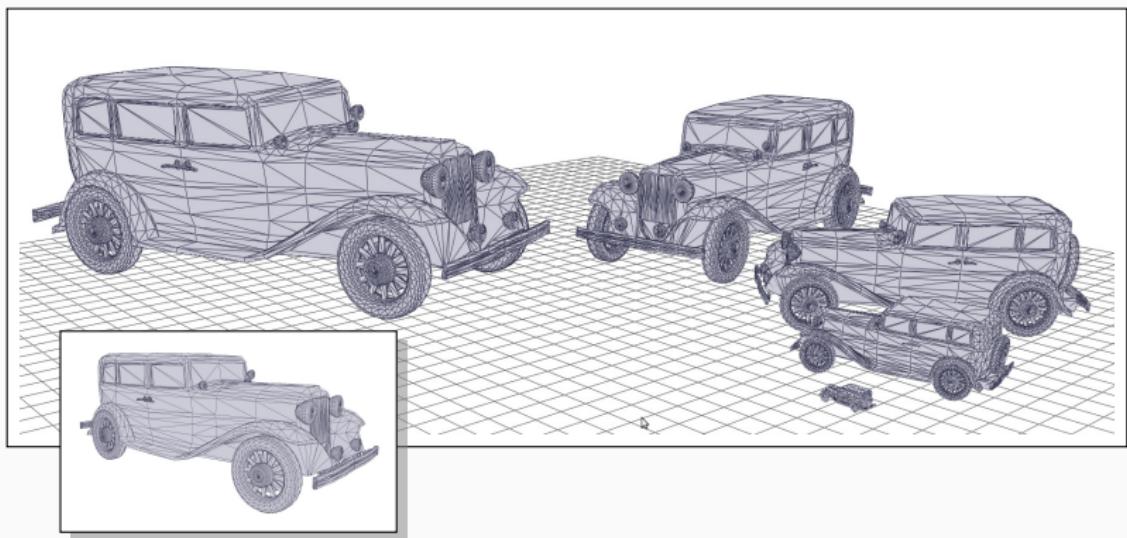
Las mallas que hemos visto en la sección anterior tienen las coordenadas de sus vértices definidas respecto de un sistema de referencia local (coordenadas maestras o locales) , pero :

- ▶ 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 definición 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.

Instanciación de una malla en una escena

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

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.

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

Sección 4. Transformaciones geométricas

Subsección 4.2.

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

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

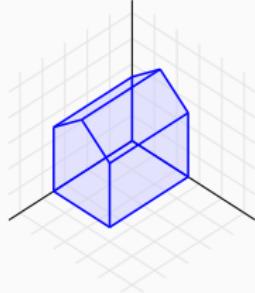
En un marco \mathcal{R} 3D las funciones y la matriz de transformación son:

$$\left. \begin{array}{rcl} x' & = & x + t_x w \\ y' & = & y + t_y w \\ z' & = & z + t_z w \\ w' & = & w \end{array} \right\} \quad \text{Tra}[d_x, d_y, d_z] \equiv \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

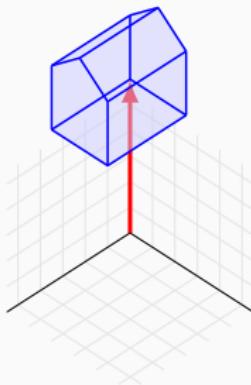
donde $(t_x, t_y, t_z, 0)^t$ son las coordenadas del \vec{t} en \mathcal{R} .

Ejemplos de traslaciones

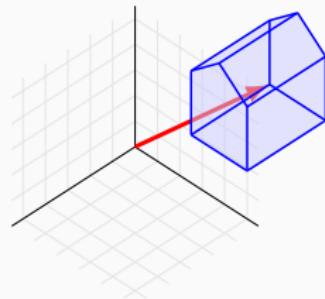
Aquí vemos a modo de ejemplo varias matrices de traslación aplicadas a los puntos de una figura sencilla, en un marco cartesiano



$$\text{Tra}[0, 0, 0]$$



$$\text{Tra}[0, 1.2, 0]$$



$$\text{Tra}[0.7, 0.6, -0.5]$$

Transformación de escalado en 3D

Un **escalado** es una transformación afín S que escala o multiplica las componentes de un vector paralelas a cada uno de los ejes de un marco arbitrario $\mathcal{R} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}]$. El punto \dot{o} no varía. Los factores de escala son (s_x, s_y, s_z) . Por tanto, se define:

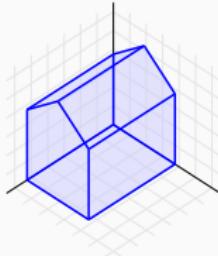
- ▶ Para puntos: $S(\vec{p}) \equiv \dot{o} + S(\vec{p} - \dot{o})$
- ▶ Para vectores: $S(c_x \vec{x} + c_y \vec{y} + c_z \vec{z}) \equiv s_x c_x \vec{x} + s_y c_y \vec{y} + s_z c_z \vec{z}$

Por tanto, las funciones y la matriz (expresadas en coordenadas de \mathcal{R}) son:

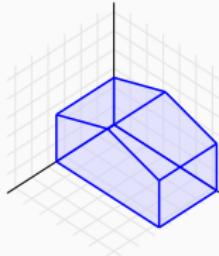
$$\left. \begin{array}{rcl} x' & = & e_x x \\ y' & = & e_y y \\ z' & = & e_z z \\ w' & = & w \end{array} \right\} \quad \text{Esc}[s_x, s_y, s_z] \equiv \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Ejemplos de transformaciones de escalado

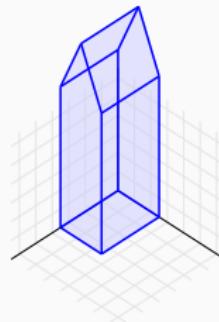
Aquí vemos varios ejemplos de la transformación de escalado.



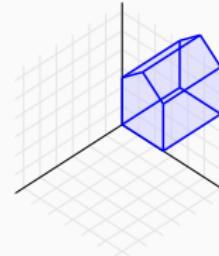
Esc[1.5, 1.5, 1.5]
uniforme
 $e_x = e_y = e_z$



Esc[2.5, 1, 1]
no uniforme
 $e_x \neq e_y = e_z$



Esc[1, 3, 1]
no uniforme
 $e_y \neq e_x = e_z$



Esc[1, 1, -1]
espejo
 $e_z < 0$

Las transformaciones de escalado no uniforme no conservan los ángulos, pero los escalados uniformes sí lo hacen.

Transformaciones de cizalla

Una transformación de cizalla (*shear*) C es como una traslación en la dirección de un eje de un marco $\mathcal{R} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}]$, pero usando una distancia proporcional (según un factor a) a la componente paralela a otro eje. El punto \dot{o} no varía. A modo de ejemplo, definimos la cizalla que desplaza X en proporción a Y:

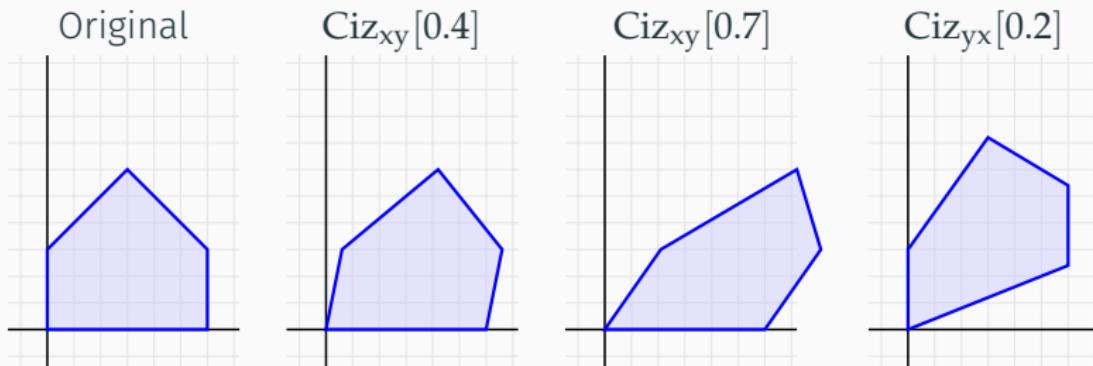
- ▶ Para puntos: $C(\vec{p}) \equiv \dot{o} + C(\vec{p} - \dot{o})$
- ▶ Para vectores: $C(c_x \vec{x} + c_y \vec{y} + c_z \vec{z}) \equiv (c_x + ac_y) \vec{x} + c_y \vec{y} + c_z \vec{z}$

Hay 6 posibles cizallas en 3D como esta, aquí vemos las funciones y matriz correspondiente a la definición anterior en coords. de \mathcal{R} :

$$\left. \begin{array}{rcl} x' & = & x + ay \\ y' & = & y \\ z' & = & z \\ w' & = & w \end{array} \right\} \quad \text{Ciz}_{xy}[a] \equiv \begin{pmatrix} 1 & a & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Ejemplos de transformaciones de cizalla.

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

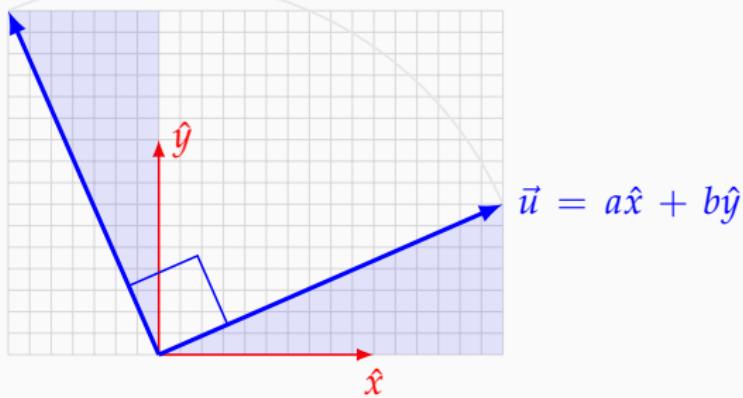


Las transformaciones de cizalla no conservan los ángulos ni las longitudes.

Rotaciones de 90° a izquierdas en 2D

En 2D definimos una transformación afín P tal que, aplicada a un vector $\vec{u} = \mathcal{C}(a, b)$, produce otro vector $P(\vec{u}) = C(-b, a)$ perpendicular a \vec{u} (girado 90° a izquierdas).

$$P(\vec{u}) = -b\hat{x} + a\hat{y}$$



aquí $\mathcal{C} = [\hat{x}, \hat{y}, \dot{o}]$ es un marco cartesiano cualquiera. Si se aplica P a un punto, se produce una rotación de 90° entorno al origen de \mathcal{C} .

Problemas: transformación afín P en 2D

Problema 2.7.

Demuestra que \vec{u} y $P(\vec{u})$ son siempre perpendiculares según la definición anterior (es decir, siempre $\vec{u} \cdot P(\vec{u}) = 0$).

Problema 2.8.

Describe como se podría definir una rotación hacia la derecha (en el sentido de las agujas del reloj) en lugar de a izquierdas.

Problema 2.9.

Demuestra que la transformación afín P (cuando se aplica a vectores, no a puntos) no depende del marco cartesiano \mathcal{C} con respecto al cual expresamos las coordenadas (a, b) (en el caso de aplicarla a puntos, la rotación de 90° es entorno al punto origen \mathbf{o} de \mathcal{C}).

Rotaciones de 90° en 3D

En 3D definimos una transformación afín similar, pero ahora hay infinitos vectores perpendiculares a \vec{u} (todos los que están en el plano perpendicular a \vec{u}).

- ▶ Para poder determinar bien la transformación afín usamos un vector \vec{e} no nulo cualquiera (para cada \vec{e} se define una transformación distinta).
- ▶ Nombramos las transformación como $P_{\vec{e}}$ (en lugar de simplemente P) y la definimos usando el producto vectorial, así:

$$P_{\vec{e}}(\vec{u}) \equiv \vec{e} \times \vec{u}$$

- ▶ Produce un vector perpendicular a \vec{u} siempre (el único que también será perpendicular a \vec{e}), por las propiedades del producto vectorial.

Matrices para la obtención un vector perpendicular

Las matrices correspondientes a P y $P_{\vec{e}}$, definidas respecto a un marco cartesiano \mathcal{C} cualquiera, son:

- En 2D:

$$P \equiv \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- En 3D:

$$P_{\mathbf{e}} \equiv \begin{pmatrix} 0 & -e_z & e_y & 0 \\ e_z & 0 & -e_x & 0 \\ -e_y & e_x & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

donde $\mathbf{e} = (e_x, e_y, e_z, 0)^t$ son las coordenadas de \vec{e} respecto de \mathcal{C}

- Estas matrices, si se aplican a puntos, suponen rotaciones de 90° entorno al origen de \mathcal{C}

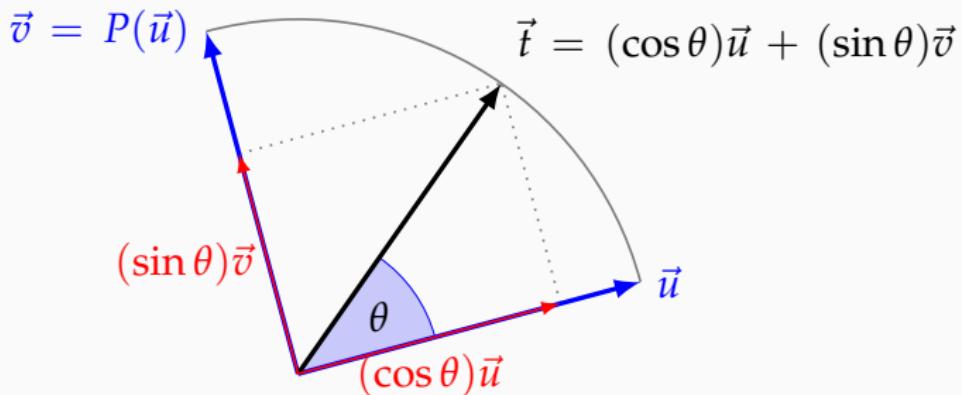
Rotaciones arbitrarias en 2D

Una transformación de rotación R_θ en 2D por un ángulo θ entorno a un punto \dot{o} transforma un vector \vec{u} o un punto \dot{p} de esta forma:

$$R_\theta(\vec{u}) = (\cos \theta)\vec{u} + (\sin \theta)P(\vec{u})$$

$$R_\theta(\dot{p}) = \dot{o} + R_\theta(\dot{p} - \dot{o})$$

El vector rotado $\vec{t} = R_\theta(\vec{u})$ es una combinación de \vec{u} y $\vec{v} = P(\vec{u})$:



Matriz de rotación arbitraria en 2D

Para obtener la matriz de rotación $\text{Rot}[\theta]$, relativa a un marco cartesiano cualquiera \mathcal{C} , usamos la matriz identidad I y la matriz P . Si \mathbf{u} son las coordenadas homogéneas de un vector en \mathcal{C} , se cumple:

$$\begin{aligned}\text{Rot}[\theta] \mathbf{u} &= (\cos \theta) \mathbf{u} + (\sin \theta) P \mathbf{u} = (\cos \theta) I \mathbf{u} + (\sin \theta) P \mathbf{u} \\ &= [(\cos \theta) I + (\sin \theta) P] \mathbf{u}\end{aligned}$$

Por tanto, deducimos como es la matriz de rotación:

$$\text{Rot}[\theta] \equiv (\cos \theta) I + (\sin \theta) P \equiv \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

y las expresiones:

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

Ejemplo de rotación arbitraria en 2D

Aquí vemos el ejemplo de una rotación por un ángulo θ (mayor que cero)



Las rotaciones son **transformaciones rígidas**: conservan los ángulos y las distancias.

Problema: invarianza ante rotaciones 2D

Problema 2.10.

Demuestra que el producto escalar de vectores en 2D es invariante por rotación, es decir, que para cualquier ángulo θ y vectores \vec{a} y \vec{b} se cumple:

$$R_\theta(\vec{a}) \cdot R_\theta(\vec{b}) = \vec{a} \cdot \vec{b}$$

(usa las coordenadas de \vec{a} y \vec{b} en un marco cartesiano cualquera)

Problema 2.11.

Demuestra que en 2D las rotaciones no modifican la longitud de un vector, es decir, que para cualquier ángulo θ y vector \vec{v} , se cumple:

$$\| R_\theta(\vec{v}) \| = \| \vec{v} \|$$

Transformaciones de rotación elementales en 3D.

En un espacio afín 3D consideramos un marco cartesiano

$\mathcal{C} = [\hat{x}, \hat{y}, \hat{z}, \dot{o}]$ y una transformación de rotación de θ radianes, con eje en un vector \vec{e} no nulo.

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

$$\text{Rot}_x[\theta] \equiv \text{Rot}[\theta, (1, 0, 0)]$$

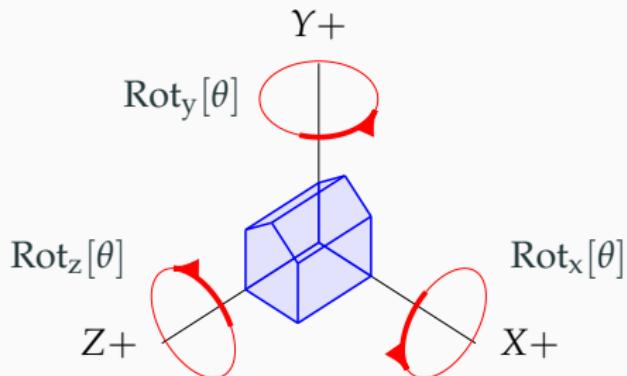
$$\text{Rot}_y[\theta] \equiv \text{Rot}[\theta, (0, 1, 0)]$$

$$\text{Rot}_z[\theta] \equiv \text{Rot}[\theta, (0, 0, 1)]$$

Las rotaciones elementales en 3D se pueden definir como las rotaciones arbitrarias en 2D, afectando a dos coordenadas y dejando invariantes la tercera.

Transformaciones de rotación elementales en 3D.

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



- Son rotaciones **siempre en sentido anti-horario** (para $\theta > 0$ cuando se mira hacia el origen desde la rama positiva del eje de giro).

Expresiones de las rotaciones elementales

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

$$\text{Rot}_x[\alpha]$$

$$x' = x$$

$$y' = cy - sz$$

$$z' = sy + cz$$

$$\text{Rot}_y[\alpha]$$

$$x' = cx + sz$$

$$y' = y$$

$$z' = -sx + cz$$

$$\text{Rot}_z[\alpha]$$

$$x' = cx - sy$$

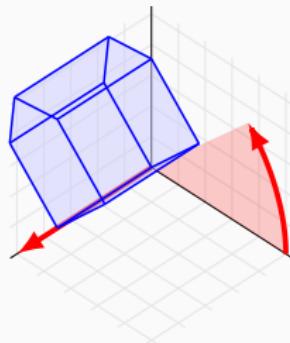
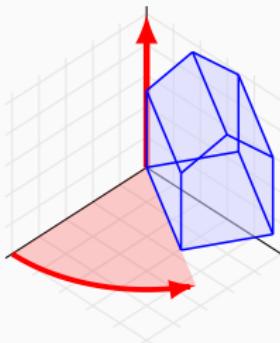
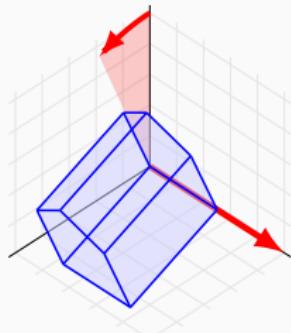
$$y' = sx + cy$$

$$z' = z$$

$$\text{Rot}_x[23^\circ]$$

$$\text{Rot}_y[60^\circ]$$

$$\text{Rot}_z[45^\circ]$$



Problemas: invarianza ante rotaciones elementales 3D

Problema 2.12.

Demuestra que el producto escalar de vectores en 3D es invariante por rotaciones elementales (usa tu solución al problema 10)

Problema 2.13.

Demuestra que las rotaciones elementales en 3D no modifican la longitud de un vector (usa tu solución al problema 11)

Problema 2.14.

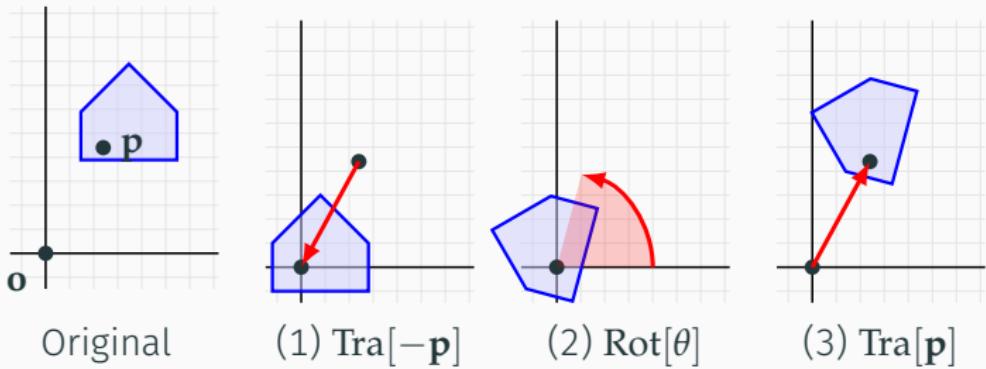
Demuestra que el producto vectorial de dos vectores rota igual que lo hacen esos dos vectores, es decir, que para cualquiera dos vectores \vec{a} y \vec{b} y un ángulo θ , se cumple:

$$R_\theta(\vec{a} \times \vec{b}) = R_\theta(\vec{a}) \times R_\theta(\vec{b})$$

Rotaciones sobre puntos arbitrarios en 2D y 3D

En un marco cartesiano \mathcal{C} , la matriz de rotación entorno a un punto \dot{p} arbitrario, distinto del origen \dot{o} del marco, se consigue componiendo (1) traslación al origen de \mathcal{C} (por el vector $\dot{o} - \dot{p}$), (2) rotación por θ (en torno al origen \dot{o}) y (3) traslación inversa (por $\dot{p} - \dot{o}$). Por tanto:

$$\text{Rot}[\theta, \mathbf{p}] = \text{Tra}[\mathbf{p}] \cdot \text{Rot}[\theta] \cdot \text{Tra}[-\mathbf{p}]$$



(donde \mathbf{p} son las coords. de \dot{p} en \mathcal{C})

Rotaciones de eje arbitrario en 3D (1/2)

En 3D una rotación R_θ tiene como **eje de rotación** una línea que pasa por \hat{o} , paralela a un vector \hat{e} (lo suponemos de longitud unidad).

- ▶ Para un punto \dot{p} se define en términos de la rotación de vectores:

$$R_\theta(\dot{p}) = \dot{o} + R_\theta(\dot{p} - \dot{o})$$

- ▶ Si \vec{u} es un vector perpendicular a \hat{e} , entonces \vec{u} rota igual que en 2D, pero en el plano perpendicular a \hat{e} . Este plano es generado por \vec{u} y otro vector \vec{v} , perpendicular a \hat{e} y \vec{u} , por tanto:

$$R_\theta(\vec{u}) = (\cos \theta)\vec{u} + (\sin \theta)\vec{v}$$

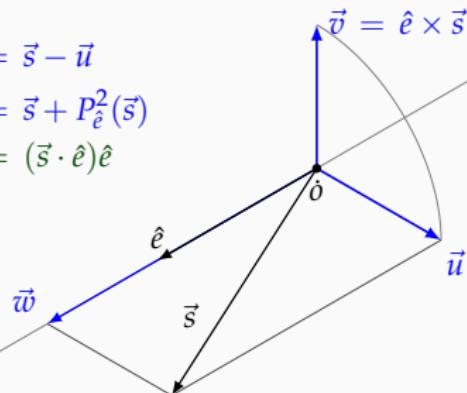
donde \vec{v} se obtiene mediante la transformación $P_{\hat{e}}$ (prod. vect.):

$$\vec{v} \equiv \hat{e} \times \vec{u} = P_{\hat{e}}(\vec{u})$$

Rotaciones de eje arbitrario en 3D (2/2)

Si \vec{s} es un vector cualquiera (no necesariamente perpendicular a \hat{e}) descomponemos \vec{s} en dos componentes: una (\vec{w}) es paralela a \hat{e} y otra (\vec{u}) es perpendicular a \hat{e} . El vector \vec{w} no rota, mientras que \vec{u} lo hace como antes. Por tanto:

$$R_\theta(\vec{s}) = \vec{w} + (\cos \theta)\vec{u} + (\sin \theta)\vec{v}$$

$$\begin{aligned}\vec{w} &= \vec{s} - \vec{u} \\ &= \vec{s} + P_{\hat{e}}^2(\vec{s}) \\ &= (\vec{s} \cdot \hat{e})\hat{e}\end{aligned}\quad \begin{aligned}\vec{v} &= \hat{e} \times \vec{s} = P_{\hat{e}}(\vec{s}) \\ \vec{u} &= -\hat{e} \times \vec{v} \\ &= -P_{\hat{e}}^2(\vec{s}) \\ &= \vec{s} - (\vec{s} \cdot \hat{e})\hat{e}\end{aligned}$$


Los vectores \vec{u}, \vec{v} y \vec{w} forman un marco de referencia ortonormal.

Fórmula de Rodrigues y expresión matricial.

En un marco cartesiano \mathcal{C} cualquiera podemos escribir las coordenadas \mathbf{s}' del vector rotado, deduciendo la **fórmula de Rodrigues de la rotación**:

$$\begin{aligned}\mathbf{s}' &= \mathbf{w} + (\cos \theta)\mathbf{u} + (\sin \theta)\mathbf{v} \\&= \mathbf{w} + (\cos \theta)(\mathbf{s} - \mathbf{w}) + (\sin \theta)\mathbf{e} \times \mathbf{s} \\&= (\cos \theta)\mathbf{s} + (1 - \cos \theta)\mathbf{w} + (\sin \theta)\mathbf{e} \times \mathbf{s} \\&= (\cos \theta)\mathbf{s} + (1 - \cos \theta)(\mathbf{s} \cdot \mathbf{e})\mathbf{e} + (\sin \theta)\mathbf{e} \times \mathbf{s}\end{aligned}$$

Donde $\mathbf{e}, \mathbf{s}, \mathbf{w}, \mathbf{u}$ y \mathbf{v} son las coordenadas en \mathcal{C} de los vectores $\vec{e}, \vec{s}, \vec{w}, \vec{u}$ y \vec{v} . Usando las matrices I y $P_{\mathbf{e}}$ podemos escribir:

$$\begin{aligned}\mathbf{s}' &= \mathbf{w} + (\cos \theta)\mathbf{u} + (\sin \theta)\mathbf{v} \\&= \left(I\mathbf{s} + P_{\mathbf{e}}^2 \mathbf{s} \right) - (\cos \theta)P_{\mathbf{e}}^2 \mathbf{s} + (\sin \theta)P_{\mathbf{e}} \mathbf{s} \\&= \left[I + (\sin \theta)P_{\mathbf{e}} + (1 - \cos \theta)P_{\mathbf{e}}^2 \right] \mathbf{s}\end{aligned}$$

Matriz de rotación de eje arbitrario en 3D

De la última igualdad se deduce cual es la matriz de rotación en 3D

$$\text{Rot}[\theta, \mathbf{e}] = I + (\sin \theta) P_{\mathbf{e}} + (1 - \cos \theta) P_{\mathbf{e}}^2$$

Haciendo la composición de las matrices, obtenemos la definición explícita:

$$\text{Rot}[\theta, \mathbf{e}] \equiv \begin{pmatrix} a_{00} + c & a_{01} - s e_2 & a_{02} + s e_1 & 0 \\ a_{10} + s e_2 & a_{11} + c & a_{12} - s e_0 & 0 \\ a_{20} - s e_1 & a_{21} + s e_0 & a_{22} + c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

donde $\mathbf{e} \equiv (e_0, e_1, e_2, 0)^t$ son las coordenadas en \mathcal{C} de \vec{e} , y

$$s \equiv \sin \theta$$

$$a_{ij} \equiv (1 - c)e_i e_j$$

$$c \equiv \cos \theta$$

Resumen: matrices de transformación 3D usuales

$$\text{Esc}[s_x, s_y, s_z] = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{Tra}[d_x, d_y, d_z] = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{Ciz}_{xy}[a] = \begin{pmatrix} 1 & a & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{Rot}_x[\alpha] = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{Rot}_y[\alpha] = \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{Rot}_z[\alpha] = \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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

Transformación de normales (1/2)

Si transformamos las posiciones de todos los vértices de una malla usando una matriz $4 \times 4 A$ (matriz de modelado), también querremos transformar los vectores normales de dichos vértices, en caso de que la malla tenga normales:

- (1) Representamos las normales como vectores columna (1×3), igual que los vectores (siempre $w = 0$)
- (2) Al igual que los vectores, las normales no se ven afectadas por traslaciones, es decir, podemos considerar A como una matriz 3×3 sin términos de traslaciones.
- (3) Supongamos que un vértice (en la malla original) tiene una normal \mathbf{n} , y que el vector tangente a la superficie en el vértice es \mathbf{t} , puesto que \mathbf{t} y \mathbf{n} son perpendiculares, se cumple

$$\mathbf{t} \cdot \mathbf{n} = 0 \iff \mathbf{t}^T \mathbf{n} = 0$$

Transformación de normales (2/2)

- (4) Al transformar la superficie por A el vector tangente transformado es $A\mathbf{t}$ (los vectores tangentes son vectores libres y se transforman como cualquier otro vector).
- (5) El vector normal se transforma por una matriz 3×3 U (desconocida en principio), si queremos que la transformación preserve la perpendicularidad se debe cumplir

$$(A\mathbf{t}) \cdot (U\mathbf{n}) = 0 \iff (A\mathbf{t})^T(U\mathbf{n}) = (\mathbf{t}^T A^T)(U\mathbf{n}) = 0$$

De (5) deducimos que $\mathbf{t}^T(A^T U)\mathbf{n} = 0$, luego por (3) deducimos que $A^T U = I$, y así sabemos que U es la **inversa de la traspuesta** de A :

$$U = (A^T)^{-1} = (A^{-1})^T$$

(aplicar U a una normal de longitud 1 puede producir una normal de long. $\neq 1$).

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

Sección 4. Transformaciones geométricas

Subsección 4.3.

Representación y operaciones sobre matrices..

El tipo Matriz4f en tup-mat.h. Operaciones.

El archivo fuente `tup-mat.h` incluye el tipo `Matriz4f`, que almacena los 16 valores (`float`) de forma contigua, proporciona operaciones para acceder a una matriz y multiplicarla por otra matriz

```
#include <tup_mat.h>
....
using namespace tup_mat ;
....

// 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 ;
```

Operaciones entre matrices y tuplas

Una matriz se puede aplicar a una tupla de 4 flotantes, se puede convertir en un puntero, se puede imprimir en varias líneas, entre otras operaciones:

```
// multiplicación de matriz 4x4 por tupla de 4 floats
Tupla4f c4a, c4b ={1.0,2.0,3.5,1.0};
c4a = m2*c4b ;

// multiplicación por tuplas de 3 flotantes (añadiendo un 1)
Tupla3f c3a, c3b = {1.0,1.6,2.8};
c3a = m2 * c3b ;

// conversión a puntero a 16 flotantes (float *) (formato compatible con OpenGL)
float * pm = m ;    const float * pcm = m ;

// escritura en la salida estándar (varias líneas)
cout << m << endl ;
```

Generación de matrices usuales en IG

También podemos usar funciones C++ para calcular las matrices más usuales (incluir `matrices-tr.h`)

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

// devuelve matrices de traslación, escalado y rotación (eje arbitrario) en 3D
inline Matriz4f MAT_Ident( ) ;
inline Matriz4f MAT_Traslacion( const Tupla3f & d ) ;
inline Matriz4f MAT_Escalado( const float sx, const float sy, const float sz ) ;
inline Matriz4f MAT_Rotacion( const float ang_gra, const Tupla3f & eje ) ;
inline Matriz4f MAT_Filas( const Tupla3f & fila0, const Tupla3f & fila1,
                           const Tupla3f & fila2 ) ;

// funciones auxiliares (construir por filas o por columnas, obtener inversa o transpuesta)
Matriz4f MAT_Columnas      ( const Tupla3f colum[3] );
Matriz4f MAT_Filas         ( const Tupla3f fila[3] );
Matriz4f MAT_Inversa       ( const Matriz4f & m );
Matriz4f MAT_Transpuesta3x3( const Matriz4f & org ) ;
```

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

Sección 4. Transformaciones geométricas

Subsección 4.4.

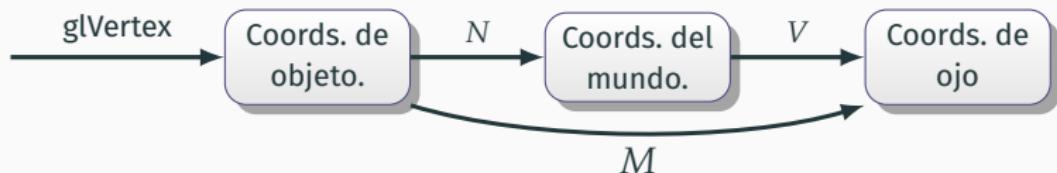
Transformaciones en OpenGL con el cauce programable.

La matriz *Modelview* en OpenGL.

Uno de los parámetros *uniform* de los shaders es una matriz $4 \times 4 M$ 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**:



Construcción de la matriz *modelview* en el caso general

En general, en cualquier aplicación OpenGL, debemos de usar funciones para generar la matriz de vista y la matriz de modelado, y usar esas matrices para configurar el cauce gráfico. Los pasos a dar para visualizar un frame son:

1. Usamos una variable M (matriz), inicialmente igual a la matriz identidad ($M := \text{Id}_e$).
2. Componemos una matriz de vista V en M (hacemos $M := M \cdot V$). Podemos usar una función como **MAT_LookAt** u otras (en 2D puede ser simplemente la matriz identidad).
3. Componemos varias matrices de transformación T_1, T_2, \dots, T_n , para ello hacemos $M := M \cdot T_i$, en orden (para i desde 1 hasta n).

Al final, queda M como $V \cdot T_1 \cdot T_2 \cdot \dots \cdot T_n$, y usamos esa matriz M para darle valor a la matriz uniform de los shaders.

Matriz Modelview en opengl3-minimo: vertex shader

En el repositorio `opengl3-minimo` hay una variable *uniform* (`u_mat_modelview`) en el *vertex shader*, de tipo matriz de 4x4 floats, que se usa para transformar todos los vértices:

```
// variable uniform: matriz de transformación de posiciones
uniform mat4 u_mat_modelview;
// atributo 0: posición del vértice en coordenadas de mundo (salida)
layout( location = 0 ) in vec3 atrib_posicion ;
....  
  
void main()
{
    ....
    // calcular las posiciones del vértice en coords. de mundo y escribimos 'gl_Position'
    gl_Position = u_mat_proyeccion *
                  u_mat_modelview * vec4( atrib_posicion, 1.0 );
}
```

El código está escrito en GLSL (usa `mat4` para matrices 4x4 y `vec3,vec4` para tuplas de flotantes)

Asignaciones a *modelview* en *opengl3-minimo*

En el código de la aplicación debemos darle valores a la matriz *modelview*

- ▶ Usamos la función **glUniformMatrix4fv** (de la familia **glUniform**).
- ▶ Se necesita la variable entera con el *localizador* del parámetro (**loc_mat_modelview**).

A modo de ejemplo, aquí vemos como asignar una matriz de traslación a la *modelview*:

```
glUniformMatrix4fv( loc_mat_modelview, 1, GL_FALSE,  
                    MAT_Traslacion( { 0.4, 0.1, -0.1 } ) );
```

En otras aplicaciones se pueden usar objetos **Matriz4f** resultado de componer varias matrices.

Visualización con *modelview* compuesta.

Suponemos que **DibujarObjeto()** envía los vértices del objeto **obj** en coordenadas maestras. El siguiente trozo de código manipula la matriz *modelview* de forma que dicho objeto se viese

- (1) rotado alrededor del eje X un ángulo igual a 45 grados, y
- (2) desplazado según el vector (5,6,7).

```
// Definir M
Matriz4f M = MAT_Traslacion( {5.0,6.0,7.0} );           // M:=Tra[5,6,7]
M = M * MAT_Rotacion( 45.0, {1.0,0.0,0.0} );          // M:=M·Rot[45º,x]

// Dar valor al uniform del shader
glUniformMatrix4fv( loc_mat_modelview, 1, GL_FALSE, M );

// Visualizar con M ≡ V·Tra[5,6,7]·Rot[45º,x]
DibujarObjeto();
```

Es usual definir funciones auxiliares para gestión de la matriz *modelview*.

Funciones auxiliares en opengl3-minimo

En el repositorio `opengl3-minimo` hay una matriz (`modelview`) y estas funciones:

- ▶ La función `resetMM` sirve para fijar la matriz `modelview` como la matriz identidad

```
void resetMM()
{
    modelview = MAT_Ident();
    ....
    glUniformMatrix4fv( loc_mat_modelview, 1, GL_FALSE, modelview );
}
```

- ▶ La función `compMM` compone una matriz `m` con la matriz `modelview` (por la derecha):

```
void compMM( const Matriz4f & m )
{
    modelview = modelview * m ;
    glUniformMatrix4fv( loc_mat_modelview, 1, GL_FALSE, modelview );
}
```

Ejemplo usando las funciones auxiliares

El ejemplo anterior puede escribirse más fácilmente usando estas funciones:

```
// Hacer modelview igual a la identidad (al inicio del frame)
resetMM();

// Dibujar otros objetos (modelview puede cambiar)
....  

// Componer las matrices en la modelview
resetMM();                                // M:=Ide
compMM( MAT_Traslacion( {5.0,6.0,7.0} ) ); // M:=Tra[5,6,7]
compMM( MAT_Rotacion( 45.0, {1.0,0.0,0.0} ) ); // M:=M · Rot[45°,x]

// Visualizar con  $M \equiv V \cdot \text{Tra}[5,6,7] \cdot \text{Rot}[45^\circ, x]$ 
DibujarObjeto();
```

La función **resetMM** debe llamarse al inicio del frame, de forma que siempre comenzamos la visualización de los objetos con la *modelview* fijada a un valor conocido.

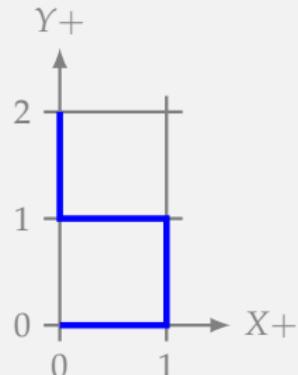
Problema: motivo básico para transformaciones

Problema 2.15.

Escribe una función llamada **gancho** para dibujar con OpenGL en modo diferido la polilínea de la figura (cada segmento recto tiene longitud unidad, y el extremo inferior está en el origen).

La función debe ser neutra respecto de la matriz *modelview*, el color o el grosor de la línea, es decir, usará la matriz *modelview*, el color y grosor del estado de OpenGL en el momento de la llamada (y no los cambia).

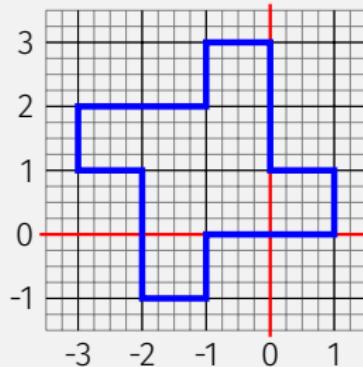
Usa la plantilla en el repositorio **opengl3-minimo** para esto.



Problema: múltiples instancias del motivo básico

Problema 2.16.

Usando (exclusivamente) la función **gancho** del problema anterior, construye otra función (**gancho_x4**) para dibujar con OpenGL, usando el **cauce fijo**, el polígono que aparece en la figura:



Usa exclusivamente **compMM** con **MAT_Traslacion** y **MAT_Rotacion**.

Problema: cálculo directo de *modelview*

Problema 2.17.

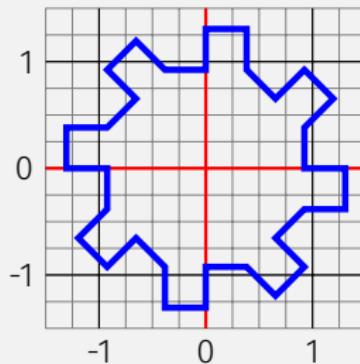
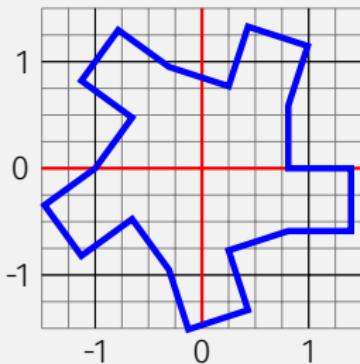
Escribe el pseudocódigo OpenGL otra función (**gancho_2p**) para dibujar esa misma figura, pero escalada y rotada de forma que sus extremos coincidan con dos puntos arbitrarios disintos \dot{p}_0 y \dot{p}_1 , puntos cuyas coordenadas de mundo son $\mathbf{p}_0 = (x_0, y_0, 1)^t$ y $\mathbf{p}_1 = (x_1, y_1, 1)^t$. Estas coordenadas se pasan como parámetro a dicha función (como **Tupla3f**)

Escribe una solución (a) acumulando matrices de rotación, traslación y escalado en la matriz *modelview* de OpenGL. Escribe otra solución (b) en la cual la matriz *modelview* se calcula directamente sin necesidad de usar funciones trigonométricas (como lo son el arcotangente, el seno, coseno, arcoseno o arcocoseno).

Problema: figura compleja

Problema 2.18.

Usa la función del problema anterior para construir estas dos nuevas figuras, en las cuales hay un número variable de instancias de la figura original, dispuestas en círculo (vemos los ejemplos para 5 y 8 instancias, respectivamente).



Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

Sección 4. Transformaciones geométricas

Subsección 4.5.

Implementación de *modelview* en la clase **Cauce..**

Métodos en la clase Cauce

Para visualización 3D, la clase **Cauce** incluye como variables de instancia las matrices de modelado (N) y vista (V) por separado. Para manipular estas matrices, se usan estos métodos

- ▶ **fijarMatrizVista(U)**

fijar V como matriz de vista (se asume que U tiene una matriz de vista, de tipo **Matriz4f**), y reinicializa la matriz de modelado (N) a la identidad. Es decir, hace $V := U$ y $N := \text{Ide}$. Se debe usar al inicio de cada frame para establecer la vista.

- ▶ **compMM(T):**

componer en la matriz de modelado actual N una matriz T por la derecha (tambien **Matriz4f**), es decir, hace: $N := N \cdot T$
Este método debe invocarse después de **fijarMatrizVista**.

Ejemplo de manipulación de *modelview* con la clase Cauce

A modo de ejemplo, este código visualiza el objeto **obj** usando una matriz de modelado obtenida como una composición de una rotación (primero), seguida de una traslación después.

```
// Definir matriz 'modelview' en el Estado de OpenGL (usando clase Cauce)
cv.cauce->fijarMatrizVista( MAT_LookAt( o, a, u )); // M := V
cv.cauce->compMM( MAT_Traslacion( {5.0,6.0,7.0} )); // M := M · Tra[5,6,7]
cv.cauce->compMM( MAT_Rotacion( 45.0, {1.0,0.0,0.0} )); // M := M · Rot[45º,x]

// Visualizar con M ≡ V · Tra[5,6,7] · Rot[45º,x]
obj->visualizarGL( cv );
```

- ▶ El objeto **cauce** forma parte de la estructura **cv** con el contexto de visualización.
- ▶ La función **MAT_LookAt** produce una *matriz de vista* (lo vemos después).

Implementación: código del *vertex shader*

En el código de prácticas se guardan por separado estas matrices:

- ▶ matriz de vista (uniform **u_mat_vista**)
- ▶ matriz de modelado (uniform **u_mat_modelado**).
- ▶ matriz de modelado de normales (unif. **u_mat_modelado_nor**).

La transformación de vértices en el *vertex shader* se hace así:

```
void main()
{
    vec4 posic_wc    = u_mat_modelado * vec4( in_posicion_occ, 1.0 ) ;
    vec3 normal_wc  = (u_mat_modelado_nor * vec4(in_normal,0.0)).xyz ;

    // calcular las variables de salida (en coords de vista)
    v_posic_ec      = u_mat_vista * posic_wc ;
    v_normal_ec     = (u_mat_vista * vec4(normal_wc,0.0)).xyz ;
    .....

    // calcular la posición del vértice en coords de dispositivo
    gl_Position     = u_mat_proyeccion * v_posic_ec ;
}
```

Implementación de la clase Cauce

La clase **Cauce** contiene (como var. de instancia):

- ▶ matriz de vista (**mat_vista**),
- ▶ matriz de modelado (**mat_modelado**)
- ▶ matriz de modelado para normales (**mat_modelado_nor**)

La implementación de **fijarMatrizVista** puede ser así:

```
void Cauce::fijarMatrizVista( const Matriz4f & nue_mat_vista )
{
    mat_vista      = nue_mat_vista ;
    mat_modelado  = MAT_Ident();
    mat_modelado_nor = MAT_Ident();

    // fijar uniforms u_mat_vista, u_mat_modelado y u_mat_modelado_nor
    glUseProgram( id_prog );
    glUniformMatrix4fv( loc_mat_vista,      1, GL_FALSE, mat_vista );
    glUniformMatrix4fv( loc_mat_modelado,   1, GL_FALSE, mat_modelado );
    glUniformMatrix4fv( loc_mat_modelado_nor, 1, GL_FALSE, mat_modelado_nor);
}
```

Componer una matriz de modelado en el cauce programable

En este caso se deben actualizar la matriz de modelado y la matriz de modelado de normales (usando la transpuesta de la inversa de la matriz a componer), y después actualizar los uniforms a su nuevo valor:

```
void Cauce::compMM( const Matriz4f & mat_comp )
{
    // actualizar matrices de modelado
    const Matriz4f
        mat_comp_nor = MAT_Transpuesta3x3( MAT_Inversa( mat_comp ) );

    mat_modelado      = mat_modelado * mat_comp;
    mat_modelado_nor = mat_modelado_nor * mat_comp_nor ;

    // fijar uniforms u_mat_modelado y u_mat_modelado_nor
    glUseProgram( id_prog );
    glUniformMatrix4fv( loc_mat_modelado,      1, GL_FALSE, mat_modelado );
    glUniformMatrix4fv( loc_mat_modelado_nor, 1, GL_FALSE, mat_modelado_nor );
}
```

Sección 5. Modelos jerárquicos. Representación y visualización..

- 5.1. Modelos Jerárquicos y grafo de escena.
- 5.2. Grafos tipo PHIGS. Ejemplo.
- 5.3. Representación de grafos.
- 5.4. Visualización de grafos en OpenGL.
- 5.5. Grafos parametrizables.

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

Sección 5. Modelos jerárquicos. Representación y visualización.

Subsección 5.1.

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 $S = \{O_1, O_2, \dots, O_n\}$)

- ▶ 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}).
- ▶ Podemos ver la matriz T_i como algo que sirve para convertir desde coordenadas relativas a \mathcal{R}_i hacia 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**.

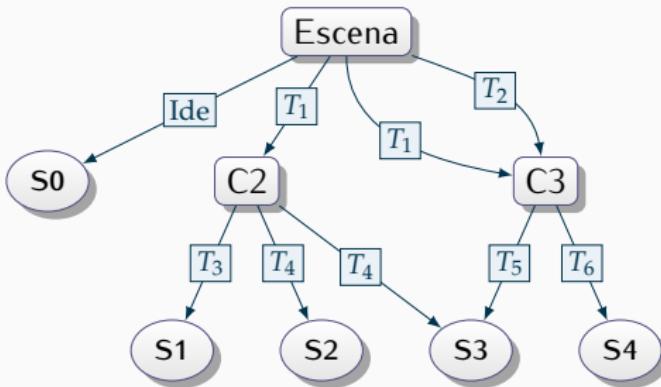
Cada objeto O_i del esquema anterior puede ser de **dos tipos** de objetos geométricos:

- ▶ **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 parente)
- ▶ 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 raíz 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).

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

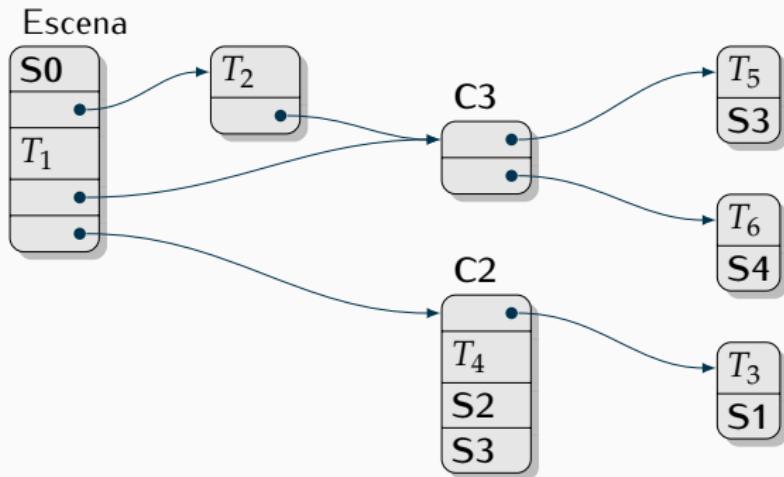
Sección 5. Modelos jerárquicos. Representación y visualización.

Subsección 5.2.

Grafos tipo PHIGS. Ejemplo..

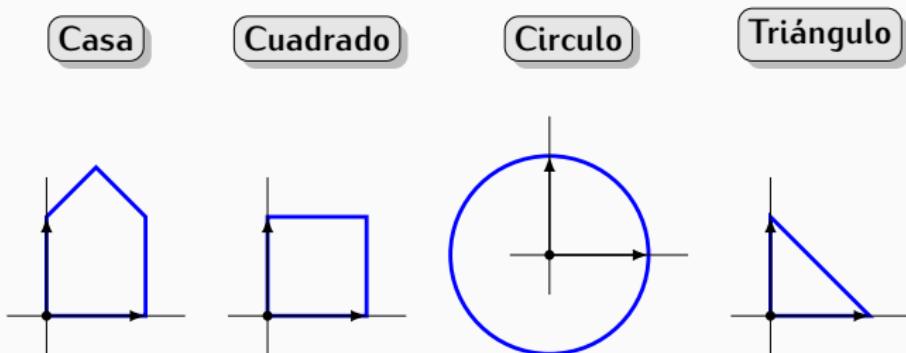
Notación inspirada en PHIGS para grafos de escena

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:



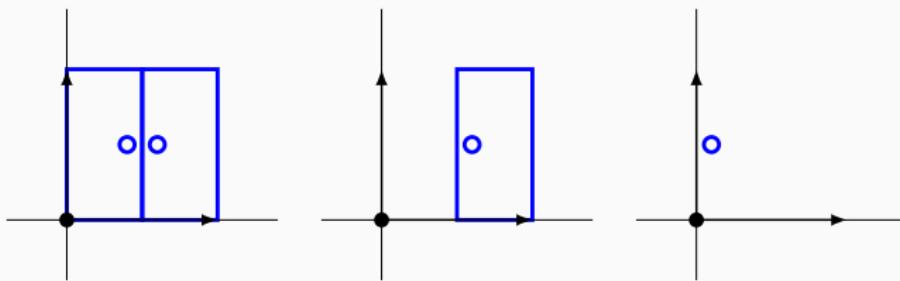
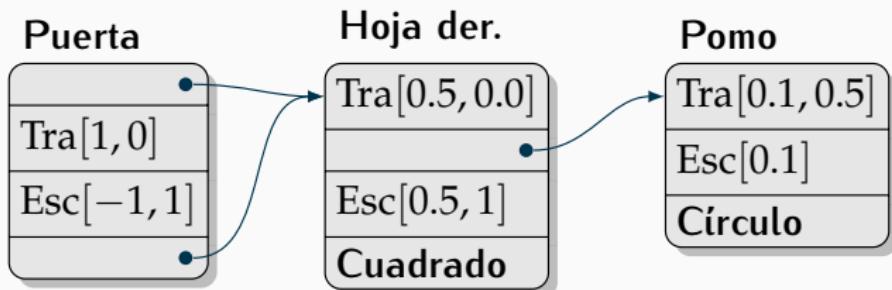
Ejemplo en 2D: objetos simples

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



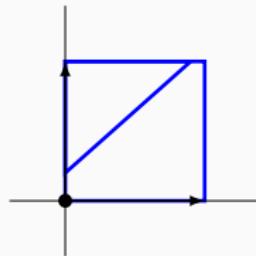
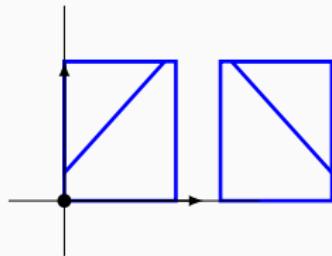
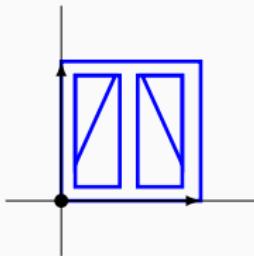
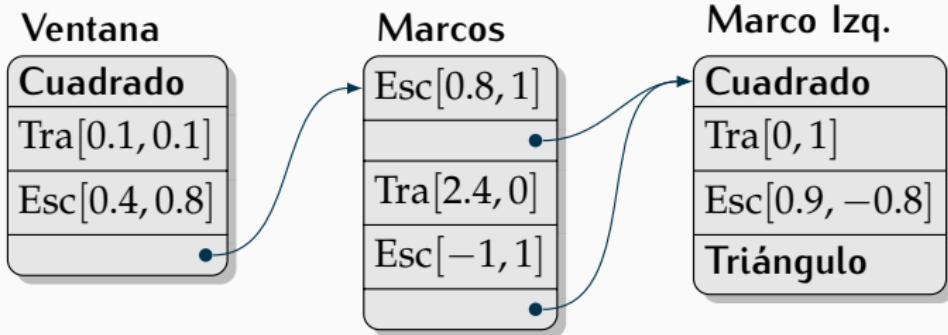
Objeto compuesto: Puerta

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



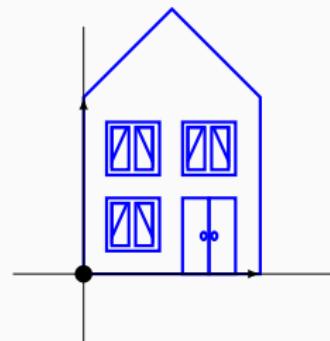
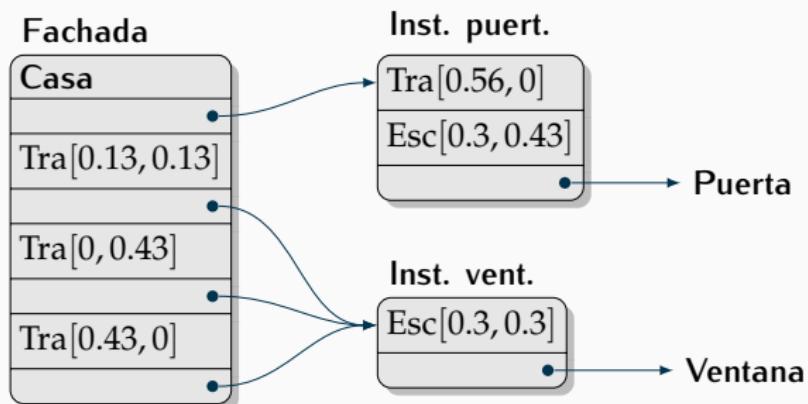
Objeto compuesto: Ventana

Una ventana hecha de cuadrados y triángulos:



Objeto compuesto: Fachada

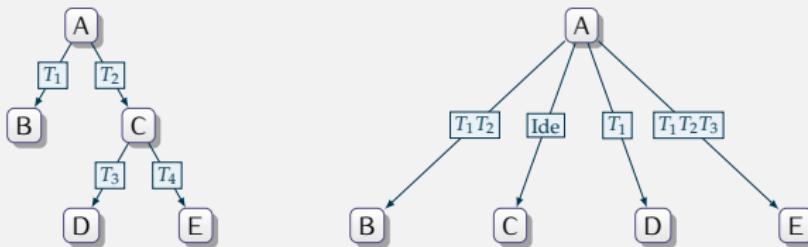
Los dos objetos compuestos creados se pueden reutilizar en un objeto de más alto nivel:



Problema: grafos de escena: arcos etiquetados versus tipo PHIGS

Problema 2.19.

Dados los dos siguientes grafos de escena sencillos:



Construye los grafos tipo PHIGS equivalentes más sencillos posible (en el sentido de menos nodos posibles). Nota: en el grafo de la derecha, hay que tener en cuenta que algunas de las transformaciones asociadas a los arcos son composiciones distintas de estas tres transformaciones: T_1, T_2 y T_3 .

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

Sección 5. Modelos jerárquicos. Representación y visualización.

Subsección 5.3.

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:

```
// tipo enumerado con los tipos de entradas de los nodos del grafo:  
enum class TipoEntNGE { objeto, transformacion, .... } ;  
  
// Entrada del nodo del Grafo de Escena  
struct EntradaNGE  
{  
    TipoEntNGE tipoE ; // tipo de entrada (enumerado)  
  
    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). Devuelve índice entrada.
    unsigned agregar( EntradaNGE * entrada ); // genérica
    // construir una entrada y añadirla (al final)
    unsigned agregar( Objeto3D * pObjeto ); // objeto (copia puntero)
    unsigned agregar( const Matriz4f & pMatriz ); // matriz (crea copia)
};
```

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

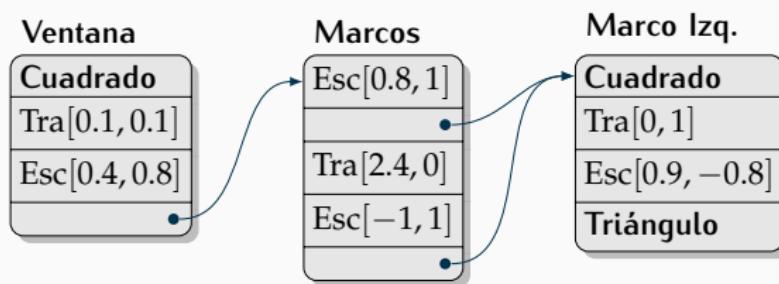
La declaración de las clases se puede hacer así:

```
// 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 ) ); // Tra[0.1,0.1]
    agregar( MAT_Escalado( 0.4,0.8,1.0 ) ); // Esc[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 ) ); // Tra[0,1]
    agregar( MAT_Escalado( 0.9,-0.8,1.0 ) ); // Esc[0.9, -0.8]
    agregar( new Triangulo );                                // Triangulo
}

Marcos::Marcos()
{
    MarcoIzq * marco_izq = new MarcoIzq ;
    agregar( MAT_Escalado( 0.8,0.1,1.0 ) ); // Esc[0.8,0.1]
    agregar( marco_izq );                            // Marco Izq.
    agregar( MAT_Traslacion( 2.4,0.0,0.0 ) ); // Tra[2.4,0]
    agregar( MAT_Escalado( -1.0,1.0,1.0 ) ); // Esc[-1,1]
    agregar( marco_izq );                            // Marco Izq.
}
```

Creación directa

También es posible crear nodos directamente, sin definir una sub-clases, el código anterior puede hacerse también así:

```
Marcos::Marcos()
{
    // crear un nuevo nodo (marco izquierdo) cmo instancia de
    NodoGrafoEscena * marco_izq = new NodoGrafoEscena() ;
    marco_izq->agregar( new Cuadrado );
    marco_izq->agregar( MAT_Traslacion( 0.0,1.0,0.0 ) );
    marco_izq->agregar( MAT_Escalado( 0.9,-0.8,1.0 ) );
    marco_izq->agregar( new Triangulo );

    // construir el objeto 'Marcos'
    agregar( MAT_Escalado( 0.8,0.1,1.0 ) );      // Esc[0.8,0.1]
    agregar( marco_izq );                          // Marco Izq.
    agregar( MAT_Traslacion( 2.4,0.0,0.0 ) ); // Tra[2.4,0]
    agregar( MAT_Escalado( -1.0,1.0,1.0 ) ); // Esc[-1,1]
    agregar( marco_izq );                          // Marco Izq.
}
```

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

Sección 5. Modelos jerárquicos. Representación y visualización.

Subsección 5.4.

Visualización de grafos en OpenGL..

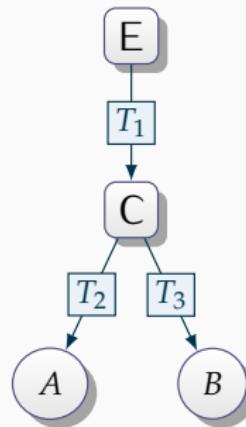
Visualización de varios objetos

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 una 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.



Visualización con OpenGL (replicando sentencias)

Para hacer la visualización con OpenGL (usando **resetMM** y **compMM**) se debe usar este código:

```
resetMM();           // M := Ide
compMM( V );        // M := M · V
compMM( T1 );      // M := M · T1
compMM( T2 );      // M := M · T2
VisualizarObjetoA(); // visualizar A con M == V · T1 · T2

resetMM();           // M := Ide
compMM( V );        // M := M · V
compMM( T1 );      // M := M · T1
compMM( T3 );      // M := M · T2
VisualizarObjetoB(); // visualizar B con M == V · T1 · T3
```

Es decir: es necesario replicar llamadas para acumular en M las matrices V y T_1 . En grafos de escena complejos:

- ▶ El código es poco legible y muy difícilmente modificable, ya que contiene líneas replicadas en muchos sitios.
- ▶ Es lento, ya que se repiten llamadas

Guardar y recuperar la matriz Modelview

Para solventar los problemas descritos, en OpenGL se suelen usar funciones auxiliares que permiten guardar la matriz modelview M y restaurarla después. En el caso del repositorio `opengl3-minimo`, usamos las funciones **pushMM** y **popMM**:

```
resetMM() ;           //  $M := \text{Id}$  (unicamente al inicio del frame)
compMM( V );          //  $M := M \cdot V$ 
compMM( T1 );        //  $M := M \cdot T_1$ 

pushMM();              //  $C := M$  (guarda una copia de  $M$  en  $C$ )
compMM( T2 );          //  $M := M \cdot T_2$ 
VisualizarObjetoA();    // visualizar  $A$  con  $M == V \cdot T_1 \cdot T_2$ 
popMM();                //  $M := C$  (restaura copia de  $M$ )

pushMM();              //  $C := M$  (guarda una copia de  $M$  en  $C$ )
compMM( T3 );          //  $M := M \cdot T_3$ 
VisualizarObjetoB();    // visualizar  $B$  con  $M == V \cdot T_1 \cdot T_3$ 
popMM();                //  $M := C$  (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 se 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
...
pushMM() ;    // P[tope] = M ; tope=tope+1
...
popMM() ;    // tope = tope-1 ; M = P[tope]
...
```

- ▶ La función **resetMM** reinicializa la pila (únicamente la invocamos al inicio del frame).
- ▶ Cada *pop* debe aparecer con igual indentación que su *push*.

Implementación de *push* y *pop*

La implementación de **pushMM** y **popMM** en opengl3-minimo es sencilla, usa **pila_modelview** (un **std::vector** de **Matriz4f**):

```
void resetMM()
{
    modelview = MAT_Ident();
    pila_modelview.clear();      // <- aquí se vacía la pila de matrices
    glUniformMatrix4fv( loc_mat_modelview, 1, GL_FALSE, modelview );
}

void pushMM()
{
    pila_modelview.push_back( modelview );
}

void popMM()
{
    assert( 0 < pila_modelview.size() ); // falla si pila vacía
    modelview = pila_modelview[ pila_modelview.size()-1 ]; // lee tope
    pila_modelview.pop_back(); // elimina tope
    glUniformMatrix4fv( loc_mat_modelview, 1, GL_FALSE, modelview );
}
```

Visualización de grafos de escena con *push* y *pop*

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

- ▶ 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()
{   pushMM();
    Casa();
    pushMM(); // inst.puerta
        compMM( MAT_Traslacion( { 0.56, 0.0, 0.0 } ) );
        compMM( MAT_Escalado( 0.3, 0.43, 1.0 ) );
        Puerta();
    popMM();
    compMM( MAT_Traslacion( { 0.13, 0.13, 0.0 } ) );
    InstVentana();
    compMM( MAT_Traslacion( { 0.0, 0.43, 0.0 } ) );
    InstVentana();
    compMM( MAT_Traslacion( { 0.43, 0.0, 0.0 } ) );
    InstVentana();
    popMM();
}
```

```
void InstVentana()
{   pushMM();
    compMM( MAT_Escalado( 0.3,0.3,1.0 ) );
    Ventana();
    popMM();
}
```

Problemas: uso de *push* y *pop* (1/2)

Problema 2.20.

Escribe una función llamada **FiguraSimple** que dibuje con OpenGL en modo diferido la figura que aparece aquí (un cuadrado de lado unidad, relleno de color, con la esquina inferior izquierda en el origen, con un triángulo inscrito, relleno del color de fondo).

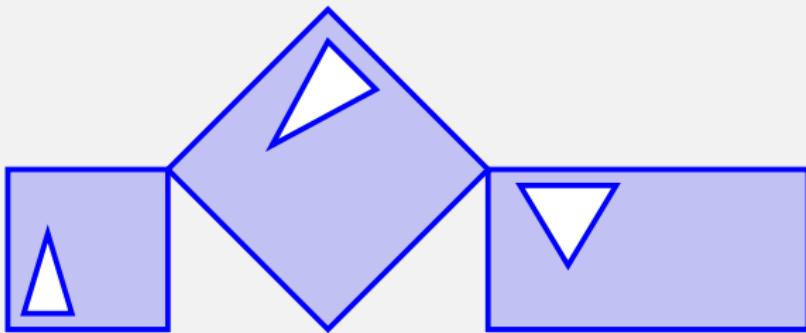


(usa el repositorio **opengl3-minimo**)

Problemas: uso de *push* y *pop* (2/2)

Problema 2.21.

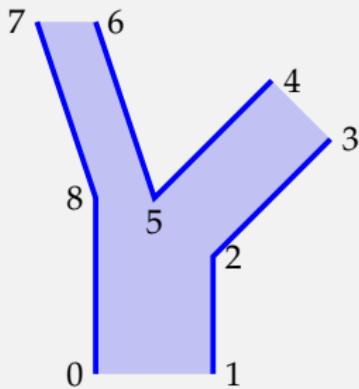
Usando exclusivamente llamadas a la función del problema 21, construye otra función llamada **FiguraCompleja** que dibuja la figura de aquí. Para lograrlo puedes usar manipulación de la pila de la matriz modelview (**pushMM** y **popMM**), junto con **MAT_Traslacion** y **MAT_Escalado**:



Problema: tronco de figura recursiv

Problema 2.22.

Escribe el código OpenGL de una función (llamada **Tronco**) que dibuje la figura que aparece a aquí. El código dibujará el polígono relleno de color azul claro, y las aristas que aparecen de color azul oscuro (ten en cuenta que no todas las aristas del polígono relleno aparecen).

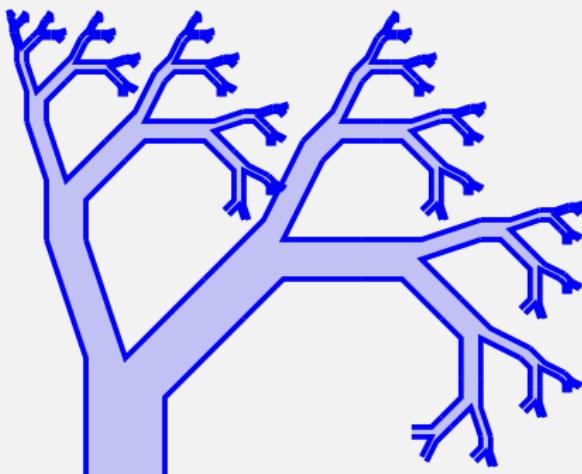


Índice	Coordenadas
0	(+0.0, +0.0)
1	(+1.0, +0.0)
2	(+1.0, +1.0)
3	(+2.0, +2.0)
4	(+1.5, +2.5)
5	(+0.5, +1.5)
6	(+0.0, +3.0)
7	(-0.5, +3.0)
8	(+0.0, +1.5)

Problema: figura recursiva.

Problema 2.23.

Escribe una función **Arbol** la cual, mediante múltiples llamadas a **Tronco** del problema 2.22, dibuje el árbol que aparece en la figura de abajo. Diseña el código usando recursividad, de forma que el número de niveles sea un parámetro modificable en dicho código (en la figura es 6)



Métodos para *push* y *pop* en la clase Cauce

La clase **Cauce**, para visualización 3D en las prácticas, incluye, como variables de instancia, dos pilas:

- ▶ una de matrices de modelado (**pila_mat_modelado**)
- ▶ otra de matrices de modelado de normales
(pila_mat_modelado_nor)

Además, existen los métodos de manipulación de estas pilas:

- ▶ método **pushMM()** para apilar una copia de ambas matrices, cada una en su pila
- ▶ método **popMM()** para restaurar las matrices usando las de los topes, y eliminarlos

Además, hay que tener en cuenta que:

- ▶ el método **fijarMatrizVista(V)** ahora vacía ambas pilas (se llama una vez al inicio del frame).

Implementación de la pila en el cauce programable

La implementación de *push* y *pop* en la clase **Cauce** sería así:

```
void Cauce::pushMM()
{ // hacer push en cada pila de las correspondiente matriz
    pila_mat_modelado.push_back( mat_modelado );
    pila_mat_modelado_nor.push_back( mat_modelado_nor );
}
```

```
void Cauce::popMM()
{ // copiar tope de cada pila en la correspondiente matriz
    const unsigned n = pila_mat_modelado.size() ; assert(0<n);
    mat_modelado      = pila_mat_modelado[n-1] ;
    mat_modelado_nor = pila_mat_modelado_nor[n-1] ;
    // eliminar tope de cada pila
    pila_mat_modelado.pop_back();
    pila_mat_modelado_nor.pop_back();
    // fijar uniforms de los shaders
    glUseProgram( id_prog );
    glUniformMatrix4fv( loc_mat_modelado,      1, GL_FALSE, mat_modelado);
    glUniformMatrix4fv( loc_mat_modelado_nor, 1, GL_FALSE, mat_modelado_nor);
}
```

Inicialización de las pilas en el cauce programable

Cuando se fija la matriz de vista (al principio de visualizar cada escena), se limpian las dos pilas de la instancia

```
void Cauce::fijarMatrizVista( const Matriz4f & nue_mat_vista )
{
    // inicializar matrices (variables de instancia)
    ...

    // inicializar pilas:
    pila_mat_modelado.clear();
    pila_mat_modelado_nor.clear();

    // fijar uniforms de los shaders
    ....
}
```

(en la práctica, si los *push* y *pop* están balanceados como deben, esto no sería necesario).

Visualización de grafos

El ejemplo de visualización de un grafos anteriores es un código específico para ese grafo:

- ▶ Ventajas: es sencillo para escenas muy sencillas o partes simples de una escena, no requiere ninguna estructura de datos en memoria.
- ▶ Inconvenientes: distintas escenas requieren distinto código, no es posible cargar un grafo de escena almacenado en un archivo en disco.

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

- ▶ Un único código sirve para visualizar cualquier grafo de escena.
- ▶ Permite visualizar grafos almacenados en archivos, creados con herramientas de diseño asistido.

El método *visualizar* en grafos.

El método **visualizarGL** dibuja recursivamente estructuras completas. Para ello usa el cauce que esté guardado en **cv**:

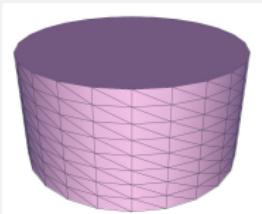
```
void NodoGrafoEscena::visualizarGL( ContextoVis & cv )
{
    // guarda modelview actual
    cv.cauce->pushMM();

    // recorrer todas las entradas del array que hay en el nodo:
    for( unsigned i = 0 ; i < entradas.size() ; i++ )
        switch( entradas[i].tipoE )
        {
            case TipoEntNGE::objeto :                      // entrada objeto:
                entradas[i].objeto->visualizarGL( cv ); // visualizar objeto
                break ;
            case TipoEntNGE::transformacion :               // entrada transf.:
                cv.cauce->compMM( *(entradas[i].matriz)); // componer matriz
                break ;
            case ....
            ....
        }
    // restaura modelview guardada
    cv.cauce->popMM() ;
}
```

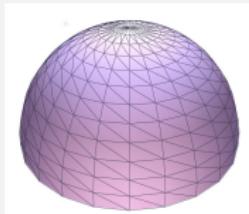
Problema: grafo PHIGS en 3D, e implementación (1/2)

Problema 2.24.

Supón que dispones de dos funciones para dibujar dos mallas u objetos simples: **Semiesfera** y **Cilindro**. La semiesfera (en coordenadas maestras) tiene radio unidad, centro en el origen y el eje vertical en el eje Y. Igualmente el cilindro tiene radio y altura unidad, el centro de la base está en el origen, y su eje es el eje Y.



Cilindro



Semiesfera



Android

(continua en la siguiente transparencia)

Problema: grafo PHIGS en 3D, e implementación (2/2)

Problema 2.24. (continuación)

Con estas dos primitivas queremos escribir el código que visualiza la figura Android, usando la plantilla de código de prácticas. Para ello:

- ▶ Diseña el grafo de escena (tipo PHIGS) correspondiente, ten en cuenta que hay objetos compuestos que se pueden instanciar más de una vez (cada brazo o pierna se puede construir con un objeto compuesto de dos semiesferas en los extremos de un cilindro).
- ▶ Escribe el código OpenGL para visualizarlo, usando una clase llamada **Android**, derivada de **NodoGrafoEscena**.

Informática Gráfica, curso 2022-23.

Teoría. Tema 2. Modelos de objetos.

Sección 5. Modelos jerárquicos. Representación y visualización.

Subsección 5.5.

Grafos parametrizables..

Modelos parametrizables

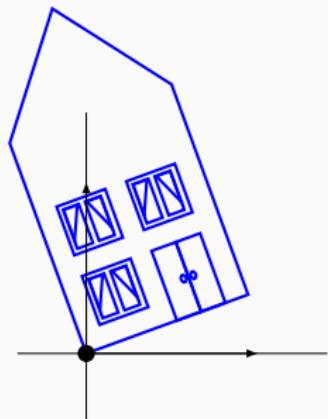
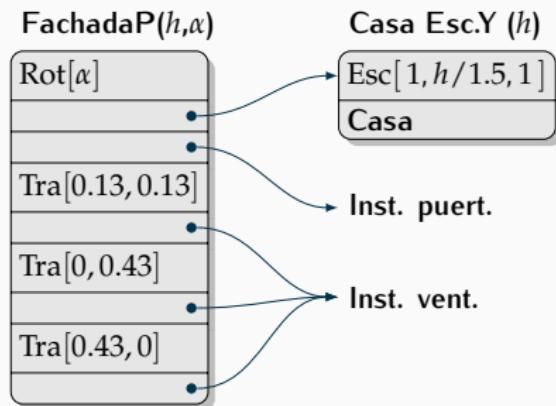
Un grafo de escena puede estar parametrizado respecto de ciertos valores reales:

- ▶ Dichos valores pueden 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 directa de FachadaP

Se añaden parámetros a las funciones:

```
void FachadaP( float h, float alpha )
{
    pushMM();
    compMM( MAT_Rotacion( alpha, { 0.0,0.0,1.0 } );
    CasaEscY( h );
    InstPuerta() ;
    compMM( MAT_Traslacion( { 0.13, 0.13, 0.0 } );
    InstVentana() ;
    compMM( MAT_Traslacion( { 0.0, 0.43, 0.0 } );
    InstVentana() ;
    compMM( MAT_Traslacion( { 0.43, 0.0, 0.0 } );
    InstVentana() ;
    popMM();
}
void CasaEscY( float h )
{
    pushMM();
    compMM( MAT_Escalado( 1.0, h/1.5, 1.0 ) );
    Casa();
    popMM();
}
```

Representacion de grafos parametrizables en memoria

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

- ▶ Un método para cambiar el valor de cada parámetro.
- ▶ Un puntero a la matriz o matrices afectadas por el parámetro.

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

```
class FachadaP : public NodoGrafoEscena
{
protected: // punteros a matrices
    Matriz4f * pm_rot_alpha = nullptr,
               * pm_escy_h   = nullptr ;
public:
    FachadaP( const float h_inicial, const float alpha_inicial );
    void fijarH( const float h_nuevo ) ;
    void fijarAlpha( const float alpha_nuevo );
} ;
```

Métodos para cambiar un parámetro

Estos dos métodos recalcularan las matrices correspondientes del grafo, en función del nuevo valor de un parámetro:

```
void FachadaP::fijarAlpha( const float alpha_nuevo )
{
    *pm_rot_alpha = MAT_Rotacion( alpha_nuevo, { 0.0, 0.0, 1.0 } );
}
void FachadaP::fijarH( const float h_nuevo )
{
    *pm_escy_h = MAT_Escalado( 1.0, h_nuevo/1.5, 1.0 );
}
```

Constructor de un nodo parametrizado

El constructor puede quedar así:

```
FachadaP::FachadaP( const float h_inicial, const float alpha_inicial )
{
    // crear sub-nodo tipo Casa escalada en Y (casa_ey)
    NodoGrafoEscena * casa_ey = new NodoGrafoEscena() ;
    unsigned ind1 = casa_ey->agregar(MAT_Escalado(1.0,h_inicial/1.5,1.0));
    casa_ey->agregar( new Casa() );

    // crear inst. de ventana y añadir entradas de FachadaP
    Objeto3D * inst_ventana = new InstVentana();
    unsigned ind2 = agregar( MAT_Rotacion( alpha_inicial, {0,0,1} ) );
    agregar( casa_ey );
    agregar( new InstPuerta() );
    agregar( MAT_Traslacion({0.13,0.13,0.13})); agregar( inst_ventana );
    agregar( MAT_Traslacion({0.0,0.43,0.0}));   agregar( inst_ventana );
    agregar( MAT_Traslacion({0.43,0.0,0.0}));   agregar( inst_ventana );

    // guardar los punteros a las matrices que dependen de los parámetros:
    pm_escy_h      = casa_ey->leerPtrMatriz( ind1 ) ;
    pm_rot_alpha = leerPtrMatriz( ind2 ) ;
}
```

Lectura de punteros a matrices

El método **leerPtrMatriz** de la clase **NodoGrafoEscena** devuelve el puntero a una matriz en una de sus entradas, dado el índice de la entrada

```
Matriz4f * NodoGrafoEscena::leerPtrMatriz( const unsigned indice )
{
    assert( indice < entradas.size() );
    assert( entradas[indice].tipo == TipoEntNGE::transformacion );
    assert( entradas[indice].matriz != nullptr );

    return entradas[indice].matriz ;
}
```

- ▶ Se comprueba que el índice corresponde a una entrada que contiene un puntero no nulo a una matriz.
- ▶ El índice se obtiene como valor devuelto por **agregar** (normalmente se ignora).

Problema: grafo 3D parametrizado e implementación

Problema 2.25.

Escribe una segunda versión del grafo de escena del problema 2.24, de forma que las transformaciones estén parametrizadas por dos valores reales (α y β) que expresan el ángulo de rotación del brazo izquierdo y derecho (respectivamente), en torno al eje que pasa por los centros de las dos semiesferas superiores de los brazos.

Asimismo, habrá otro parámetro (ϕ) que es el ángulo de rotación de la cabeza (completa: con los ojos y antenas) entorno al eje vertical que pasa por su centro (cuando estos ángulos valen 0, el androide está en reposo y tiene exactamente la forma de la figura del problema anterior).

Escribe el código de una nueva clase (**AndroidParam**, derivada de **NodoGrafoEscena**) para visualizar el androide parametrizado de esta forma.

Fin de la presentación.