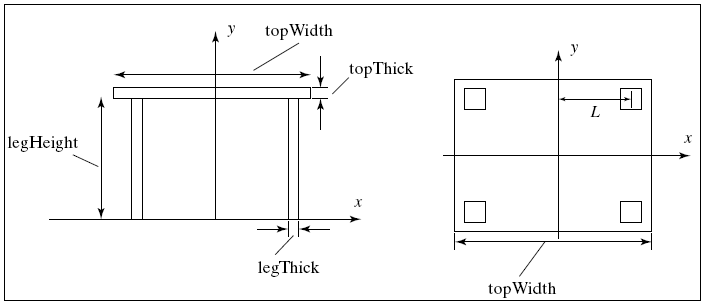
El modelo jerárquico

P. J. Martín, A. Gavilanes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

Recordatorio

- Dibujo de una mesa compuesta por un tablero y cuatro patas.




```
//<<<<<<<<<<<<<<<<<<<<<<<< table >>>>>>>>>>>>>>>>>>>>>>>
void table(double topWid, double topThick, double legThick, double legLen)
{ // draw the table - a top and four legs
    glPushMatrix(); // draw the table top
    glTranslated(0, legLen, 0);
    glScaled(topWid, topThick, topWid);
    glutSolidCube(1.0);
    glPopMatrix();
    double dist = 0.95 * topWid/2.0 - legThick / 2.0;
    glPushMatrix();
    glTranslated(dist, 0, dist);
    tableLeg(legThick, legLen);
    glTranslated(0, 0, -2 * dist);
    tableLeg(legThick, legLen);
    glTranslated(-2 * dist, 0, 2*dist);
    tableLeg(legThick, legLen);
    glTranslated(0, 0, -2*dist);
    tableLeg(legThick, legLen);
    glPopMatrix();
}
```

El tablero

Las 4 patas

Recordatorio

- El esquema general para dibujar un objeto O –como la mesa- compuesto por los objetos O1, ..., ON –en este caso, un tablero y cuatro patas- es el siguiente:

```
glPushMatrix();
   "post-multiplicar la matriz de modelado-vista actual
   por la que permite situar el sistema de coordenadas
   de forma que podamos empezar a dibujar el objeto O";
   for (int i=1; i<=N; i++)
        Oi->dibuja();
glPopMatrix();
```

- En el ejemplo de la mesa:
  - La matriz de la mesa es la identidad. Su método **`dibuja()`** es el que acabamos de ver, porque es un objeto compuesto.
  - La matriz del tablero es la que resulta de post-multiplicar, en este orden, la matriz asociada a una traslación por la asociada a una escalación. Su método **`dibuja`**`()` es directamente el método **`glutSolidCube(1.0),`** porque no es un objeto compuesto.

Modelo jerárquico

- El modelo jerárquico extiende esta idea a todos los objetos de la escena.
- Cada objeto de la escena tiene un atributo que contiene la matriz por la que es necesario post-multiplicar la de modelado-vista antes de dibujarlo.
- Este atributo es un objeto de una nueva clase **TAfin** (de **T**ransformación **Afin**) que contiene:
  - Un atributo `m` de tipo **`GLfloat[16]`** para almacenar la matriz (16 datos enumerados por columnas).
  - Operaciones que actualizan ese atributo, acumulando transformaciones:
    1. Se construye la matriz asociada a la transformación que deseamos acumular **`m1`**.
       - ✓ Si se trata de una operación básica de OpenGL podemos cargarla con el comando correspondiente de OpenGL y recuperarla con **`glGetFloatv(GL_MODELVIEW_MATRIX, m1);`**
    2. Se post-multiplica la matriz actual del atributo `m` por **`m1`**, dejando el resultado en el propio atributo: **`m= m * m1;`**
       - ✓ Podemos usar el comando de OpenGL **`glMultMatrixf(m1);`** que post-multiplica la matriz actual activada, por la matriz **`m1`**.

- Por ejemplo, la operación de traslación se implementa en **TAfin** mediante el método:

```
void TAfin::traslada(PV3D* v) {
   glMatrixMode(GL_MODELVIEW);
   glPushMatrix();
      glLoadIdentity();
      glTranslatef(v->getX(), …);
      GLfloat m1[16];
      //Dejar la matriz actual de modelado-vista en m1
      //Los 16 datos están enumerados por columnas
      glGetFloatv(GL_MODELVIEW_MATRIX, m1);
   glPopMatrix();
   postMultiplica(m1);
}
```

- La operación que post-multiplica (**m= m * m1;**) se implementa en **TAfin** así:

```
void TAfin::postmultiplica(GLfloat* m1) {
   glMatrixMode(GL_MODELVIEW);
   glPushMatrix();

      //Cargar m como matriz actual de modelado-vista
      glLoadMatrixf(m);

      //Post-multiplicar por m1
      glMultMatrixf(m1);

      //Dejar el resultado en m
      glGetFloatv(GL_MODELVIEW_MATRIX, m);

   glPopMatrix();
}
```

Modelo jerárquico

- Todos los objetos de la escena heredan de una clase **`Objeto3D`**.
- La clase **`Objeto3D`** tiene:
  - Un atributo **`TAfin* mT;`**
  - Un método virtual **`dibuja();`**
- En las escenas que manejamos, de la clase **`Objeto3D`** heredan objetos como:

Objetos no compuestos

  - La clase **`Malla`** que implementa el método **`dibuja()`** tal como hemos visto hasta ahora.
  - La clase **`ObjetoCuadrico`** de la que heredan las clases **`Cilindro`**, **`Esfera`**, **`Disco`**, **`DiscoParcial`**, y que implementan el método **`dibuja()`** a través de los objetos cuádricos de la librería `GLU`.
  - La clase **`Cubo`** que usa las funciones glutSolidCube() y glutWireCube() de la librería `GLUT` para dibujar cubos.
  - La clase **`ObjetoCompuesto`** que implementa el método **`dibuja()`** siguiendo las ideas que acabamos de ver.

Modelo jerárquico

- Una forma en que la clase **ObjetoCompuesto** puede implementar los objetos que la forman es mediante un array dinámico de objetos 3D.

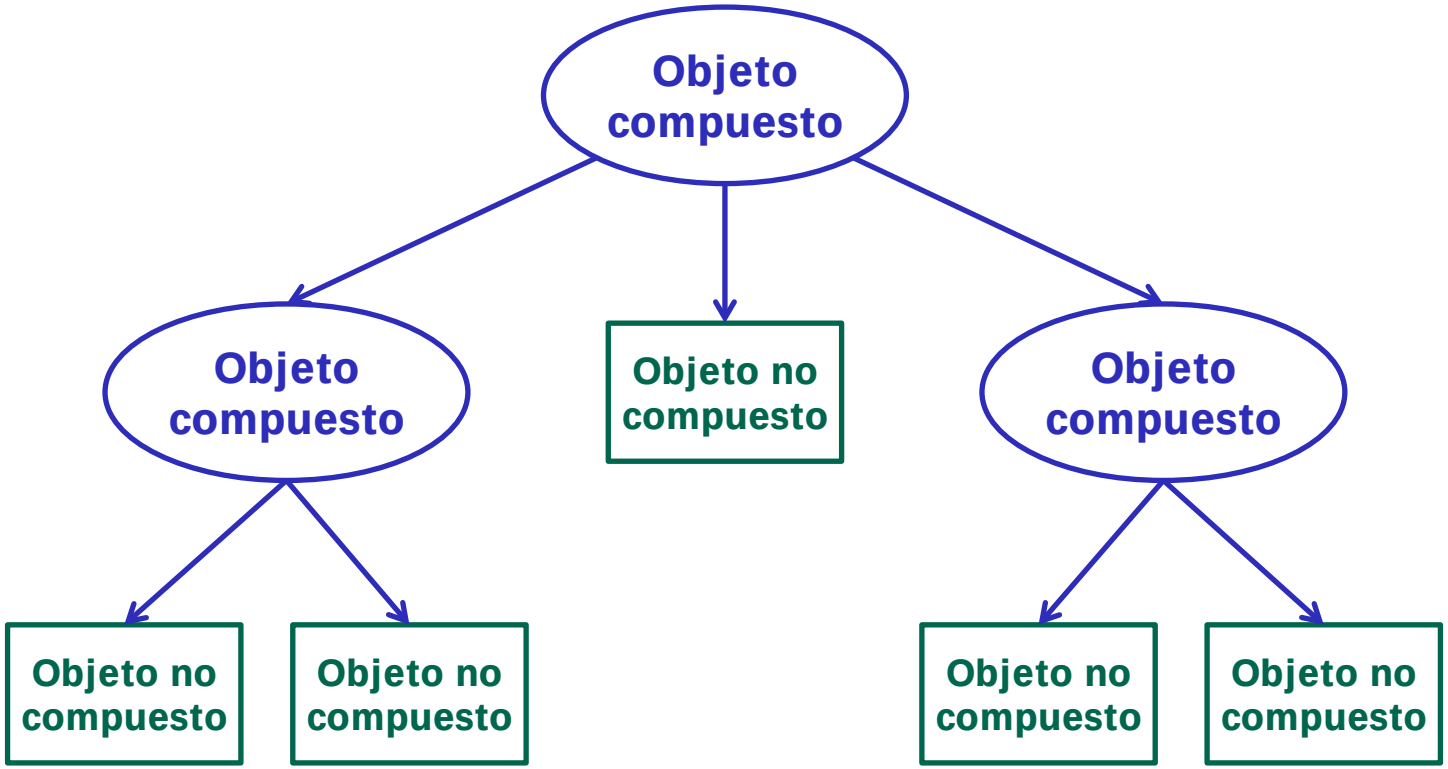
```
class ObjetoCompuesto : public Objeto3D {
    protected:
        int numHijos;
        Objeto3D** hijos;
    …
}
```

Modelo jerárquico

- La clase **ObjetoCompuesto** implementa entonces el método **dibuja()** como:

```
void ObjetoCompuesto:dibuja() {
     glMatrixMode(GL_MODELVIEW);
     glPushMatrix();
        glMultMatrix(this->mT->m);
        for (int i=0; i<numHijos; i++)
             hijo[i]->dibuja();
     glPopMatrix();
}
```

Modelo jerárquico



Modelo jerárquico

- Ventajas del modelo jerárquico:
  - Cada constituyente de un objeto compuesto debe aplicar la secuencia de transformaciones que lo coloca en el conjunto antes de dibujarse.

  En el modelo jerárquico, cada objeto guarda una única transformación que combina esa secuencia $\Longrightarrow$ se dibuja más rápidamente.

  - Durante el proceso de *culling*: si un objeto compuesto no es visible, no es necesario dibujarlo y, por tanto, tampoco es necesario dibujar los objetos que lo componen.

  En el modelo jerárquico, no es necesario codificar nada. Si el objeto compuesto no invoca el método **`dibuja()`**, sus constituyentes tampoco lo harán.

  - Si un objeto compuesto se mueve, es necesario mover todos los objetos que lo componen.

  En el modelo jerárquico, no es necesario codificar nada. Cuando el objeto compuesto se mueve, la matriz de modelado-vista se post-multiplica por la matriz de éste y el resultado sirve de base para las post-multiplicaciones que aplican sus constituyentes al dibujarse.

- Realmente, hay dos formas posibles de dibujar un objeto compuesto. Una es como aparece en las diapositivas anteriores:

```
void ObjetoCompuesto::dibuja() {
     glMatrixMode(GL_MODELVIEW);
     glPushMatrix();
            glMultMatrix(this->mT->m);
            for (int i=0; i<numHijos; i++)
                   hijo[i]->dibuja();
     glPopMatrix();
}
```

Cada objeto acumula su transformación afín.

Si se hace así, los objetos que no son compuestos (las mallas y los objetos cuádricos) también deben post-multiplicar por su matriz antes de dibujarse.

- La otra alternativa sería:

```
void ObjetoCompuesto::dibuja() {
    for(int i =0; i < numHijos; i++) {
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
            glMultMatrixf(hijos[i]->dameMatrizAfin());
            hijos[i]->dibuja();
    glPopMatrix();
    }
}
```

Los objetos compuestos acumulan la transformación afín de cada uno de sus constituyentes antes de pedir que se dibujen.

Ningún objeto acumula su propia transformación afín al dibujarse, alguien lo hará por él antes de pedir que se dibuje. ⟹ Los objetos no compuestos (mallas, objetos cuádricos…) se dibujan como hasta antes de ver el modelo jerárquico.

El coche como objeto compuesto

```
Coche :: Coche() {
   //Se añaden 7 elementos al objeto compuesto Coche
   //(1 chasis, 4 ruedas y 2 faros)
   //Por ejemplo, se añaden 4 ruedas,
   //compuestas por un cilindro y un disco
   for (int i=0; i<4; i++)
      this->introduceObjeto(new Rueda());

   //Se sitúan los 7 elementos dentro del coche
   //Por ejemplo, las ruedas 1..4
   ...
   hijos[1]->rota(90, 1, 0, 0);
   hijos[1]->traslada(…);
   ...
}
```

El coche en el modelo jerárquico