Tema 3: Representación de modelos 3D e interacción básica

- Objetivos: Tras la superación de este tema el alumno:
  - Conocerá los métodos habituales para representar escenas y modelos 3D
  - Conocerá las técnicas básicas para seleccionar e interactuar con objetos y partes de objetos de una escena 3D
  - Será capaz de escribir programas informáticos para la representación e interacción con escenas y modelos 3D

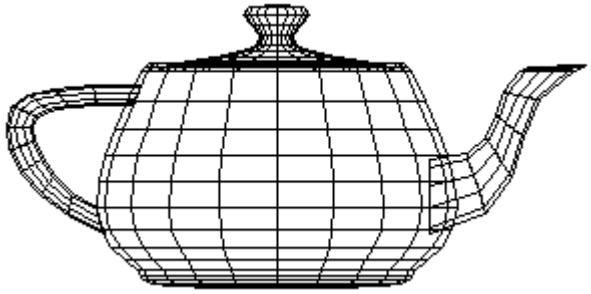
- Contenidos:
  - 3.1. Representación de primitivas básicas
  - 3.2. OpenGL: Visualización de primitivas básicas
  - 3.3. Mallas de triángulos
  - 3.4. OpenGL: Mallas de triángulos
  - 3.5. Grafos de escena
  - 3.6. OpenGL: Grafos de escena
  - 3.7. Selección e interacción con la escena
    - 3.7.1. Dispositivos de entrada y gestión de eventos
    - 3.7.2. OpenGL: Gestión de eventos
    - 3.7.3. Selección
    - 3.7.4. OpenGL: Selección mediante lista de impactos
  - 3.8. OpenGL: Práctica 3

3.1. Representación de primitivas básicas

- Una manera fácil de construir modelos complejos es partir de otros más simples, predefinidos, denominados primitivas
- Toda librería gráfica cuenta con un conjunto de primitivas gráficas parametrizadas para su utilización directa

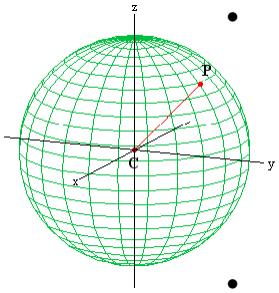
- **Poliedros**:

  - Objeto cerrado compuesto por un conjunto de caras poligonales
  - Es la representación más utilizada
  - Para su definición general hay que proporcionar los vértices y aristas que describen cada polígono, así como su vector normal (ver punto 3.3.)
  - Las librerías gráficas suelen proporcionar también una serie de funciones para visualizar poliedros predefinidos: poliedros regulares, cubos, esferas, ...



3.1. Representación de primitivas básicas

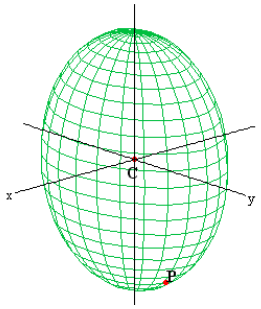
- **Cuádricas**:
  - Las cuádricas son superficies definidas mediante ecuaciones de segundo grado
  - Son superficies muy utilizadas en diversas aplicaciones de la informática gráfica, especialmente en CAD
  - Las cuádricas suelen utilizarse como primitivas para la construcción de objetos complejos. Suelen ser de especial interés las esferas y los elipsoides



  - Esfera:
    - Una esfera de radio r centrada en el origen es el conjunto de puntos que verifica:

$$x^2 + y^2 + z^2 = r^2$$

  - Elipsoide:
    - Se puede definir como una extensión de la definición de esfera, donde se proporcionan tres radios diferentes, r_x, r_y y r_z, uno en la dirección de cada eje:

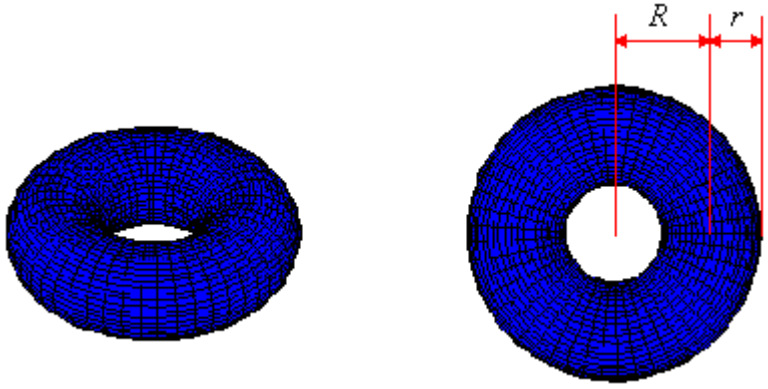


$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1$$

- **Cuádricas**:

- Toro:
  - Un toro se obtiene mediante la rotación de una circunferencia sobre un eje coplanar que no la intersecte. Para definirlo hay que dar la distancia del centro de la circunferencia al eje de rotación (radio axial) y el radio de la circunferencia
  - Los puntos pertenecientes a la superficie del toro son aquellos que verifican:

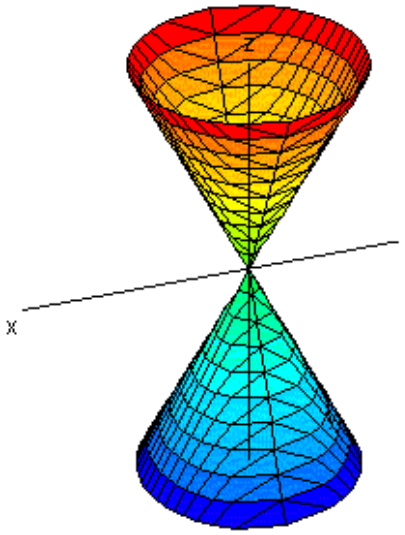
$$\left(\sqrt{x^2+y^2}-r_{axial}\right)^2+z^2=r^2$$

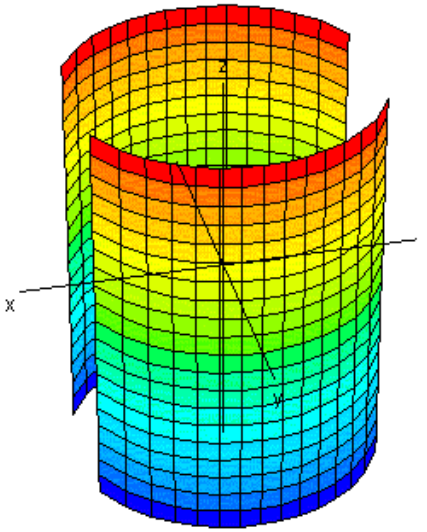


- **Cuádricas**:

- Cono:
  - Un cono circular centrado en el origen es una cuádrica degenerada cuyos puntos verifican: $x^2 + y^2 = z^2$

- Cilindro:
  - Un cilindro circular de radio r centrado en el origen es una cuádrica degenerada cuyos puntos verifican: $x^2 + y^2 = r$

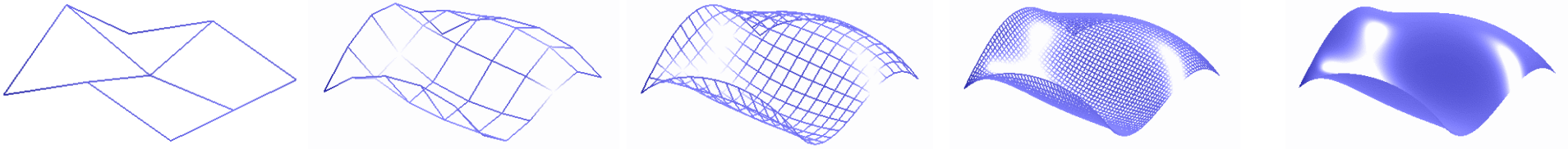




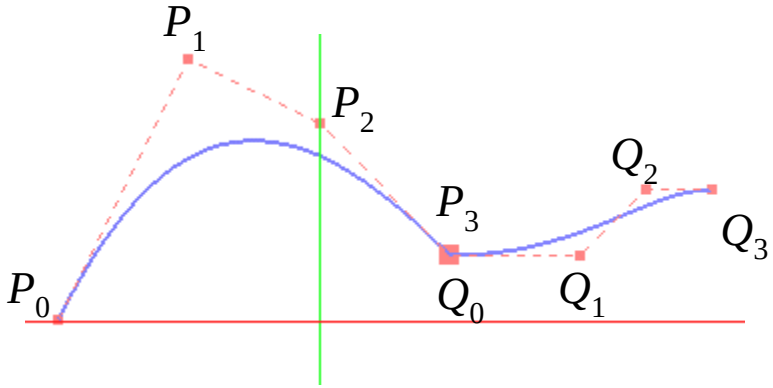
Trabajo

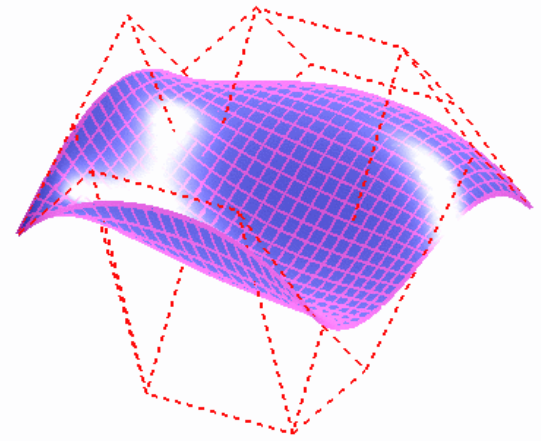
- **Superficies B-spline**:

- La representación general de **superficies curvas** como aproximaciones poligonales presenta varios problemas:
  - a mayor precisión, mayor número de triángulos necesarios (mayor uso de memoria)
  - la manipulación interactiva puede llegar a ser muy tediosa y complicada



- Una manera alternativa de construir superficies es la utilización de funciones paramétricas $f(u, v)$ de grado mayor de uno.
- Lo normal es expresar $f(u, v)$ como un **polinomio** en función de una serie de puntos de control, la superficie se calcula como un promedio de esos puntos. **Las superficies Bézier y B-spline** se utilizan mucho, fundamentalmente en entornos CAD y modelado de forma libre



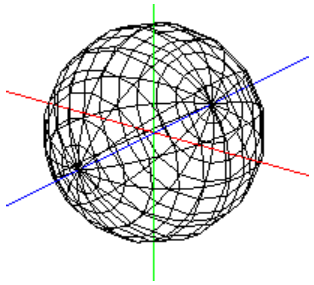


3.2. OpenGL: Visualización de primitivas básicas

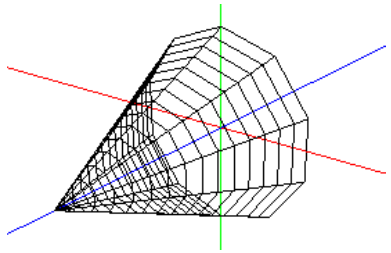
- OpenGL proporciona funciones para visualizar:
  - poliedros de forma general y mallas de polígonos (ver punto 3.4)
  - **poliedros regulares**:
    - todas las caras son polígonos regulares iguales
    - todas las aristas y ángulos entre aristas y caras son iguales
    - los poliedros se pueden visualizar en modo alambre o con relleno de polígonos (la visualización dependerá de la iluminación y las propiedades de material)
    - todos los poliedros se visualizan centrados en el origen de coordenadas
    - Tetraedro:
      - `glutWireTetrahedron();` y `glutSolidTetrahedron();`
      - la distancia del centro del tetraedro a los vértices es la raíz cuadrada de 3
    - Cubo:
      - `glutWireCube(arista);` y `glutSolidCube(arista);`
      - `arista` es la longitud de cada lado del cubo, un valor positivo `GLdouble`
    - Octaedro, dodecaedro e icosaedro:
      - `glutWire*();` con `*` = `Octahedron`, `Dodecahedron` ó `Icosahedron`
      - `glutSolid*();`
      - la distancia del centro a los vértices es 1

3.2. OpenGL: Visualización de primitivas básicas

- OpenGL proporciona **cuádricas** en sus librerías GLUT y GLU
- Cuádricas GLUT:
  - Esfera:
    - `glutWireSphere(radio, nlong, nlat);` y `glutSolidSphere(radio, nlong, nlat);`



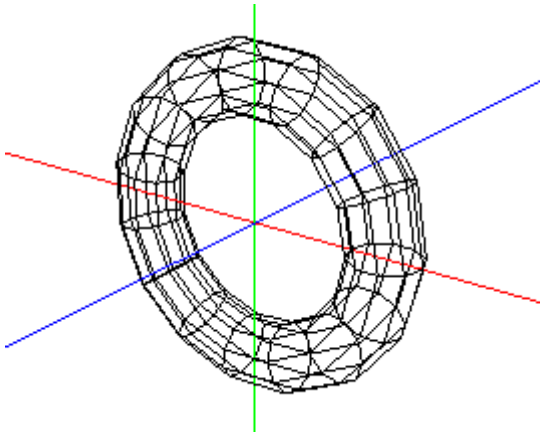
    - `radio` es un valor `GLdouble` que determina el radio de la esfera
    - `nlong` y `nlat` son valores enteros que determinan el número de líneas ortogonales que se van a utilizar para aproximar la esfera
    - la esfera se visualiza centrada en el origen de coordenadas
  - Cono:
    - `glutWireCone(rbase, altura, nlong, nlat);` y `glutSolidCone(rbase, altura, nlong, nlat);`



    - `rbase` y `altura` son valores `GLdouble` que determinan la base del cono y su altura
    - `nlong` y `nlat` son valores enteros que determinan el número de líneas ortogonales que se van a utilizar para aproximar el cono
    - el centro de la base se sitúa en el origen de coordenadas y el vértice del cono sobre el eje Z positivo

3.2. OpenGL: Visualización de primitivas básicas

- Cuádricas GLUT:
  - Toro:
    - `glutWireTorus(rcirculo, raxial, nconcen, nsec);` y `glutSolidTorus(rcirculo, raxial, nconcen, nsec);`
    - `rcirculo` y `raxial` son valores `GLdouble` que determinan el radio del círculo que define el toro y la distancia de su centro al eje Z (eje de rotación)
    - `nconcen` y `nsec` son valores enteros que determinan, respectivamente, el número de círculos concéntricos en el eje Z y el número de secciones que se van a pintar
    - el toro se centra en el origen de coordenadas con su eje axial sobre el eje Z

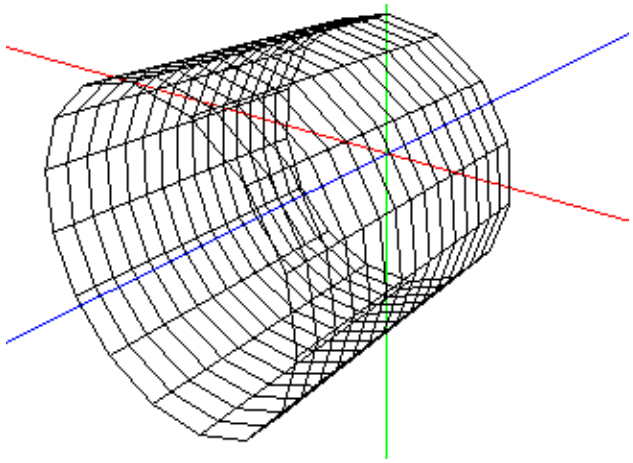


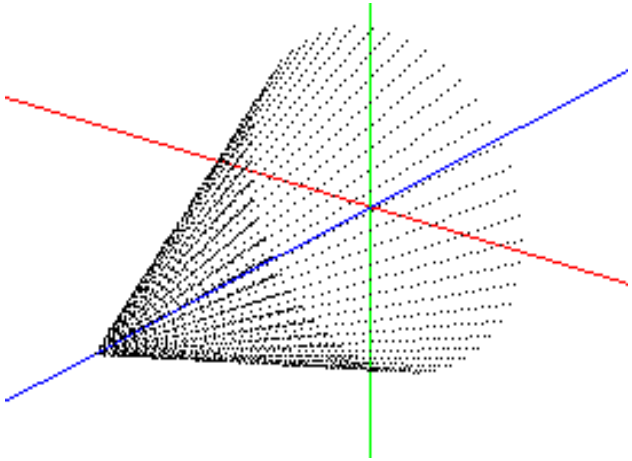
3.2. OpenGL: Visualización de primitivas básicas

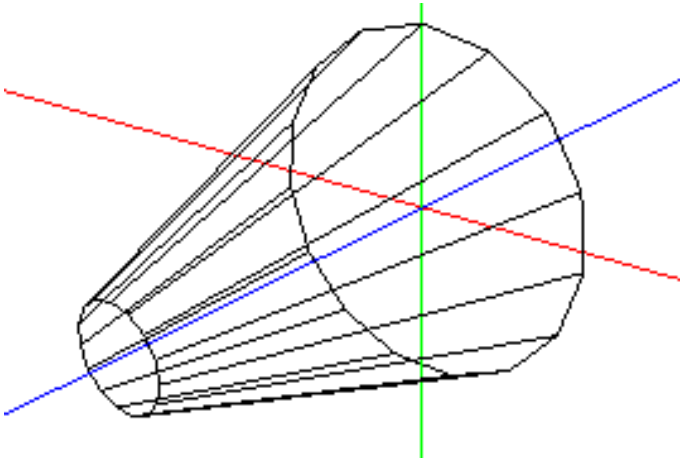
- Cuádricas GLU:
  - Los pasos para visualizar una cuádrica utilizando las funciones GLU son:
    1. Asignación de nombre a la cuádrica: `GLUquadricObj *nombre;`
    2. Activar el visualizador de cuádricas de GLU: `nombre=gluNewQuadric();`
    3. Establecer el modo de visualización:
       `gluQuadricDrawStyle(nombre, modo);`
       `GLU_LINE`, `GLU_POINT`, `GLU_SILHOUETTE` o `GLU_FILL`
    4. Establecer los valores para los parámetros de la superficie concreta:
       `gluSphere(nombre, radio, nlong, nlat);`
       `gluCylinder(nombre, rbase, rtope, altura, nlong, nlat);`
       - si `rtope=0` obtenemos un cono
       - si `rtope=rbase` obtenemos un cilindro
       - si `rtope<>rbase` obtenemos un cono truncado
       - la base está centrada en el origen y el tope se sitúa en el eje Z positivo
    5. Eliminar la cuádrica y liberar la memoria asignada:
       `gluDeleteQuadric(nombre);`

3.2. OpenGL: Visualización de primitivas básicas

- Cuádricas GLU:
  - Ejemplos de cilindros:
    - `rbase=rtope` y `GL_LINE`
    - `rtope=0` y `GL_POINT`
    - `rtope<rbase` y `GL_SILHOUETTE`

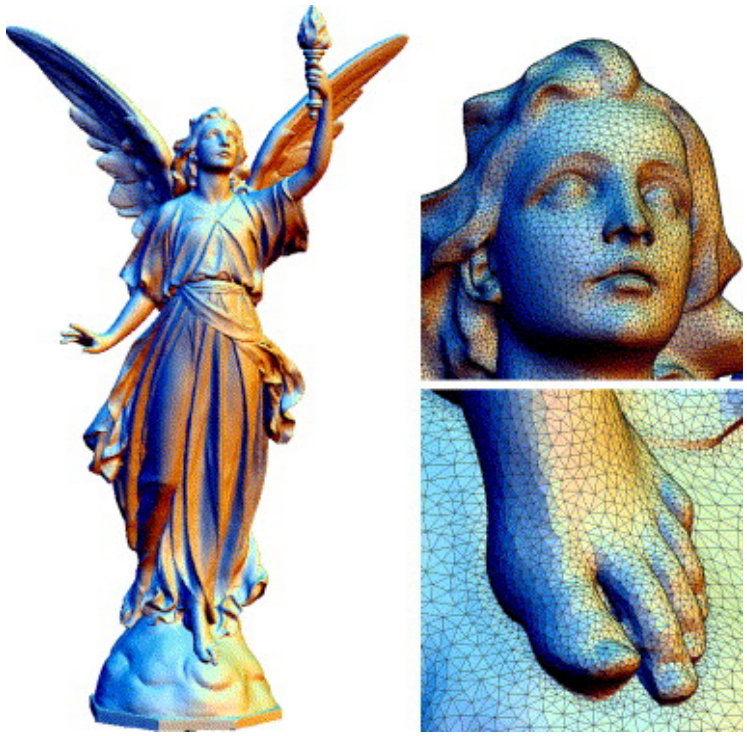




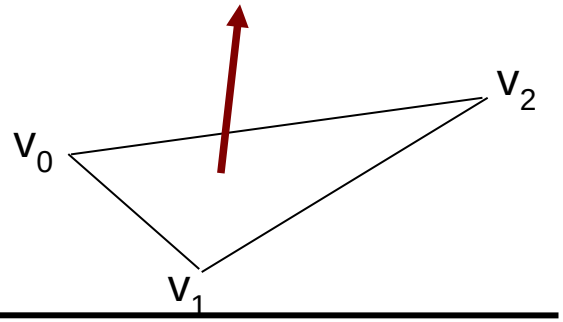


3.3. Mallas de triángulos

- Una típica manera de representar una superficie es mediante un conjunto de polígonos
- Una malla de polígonos es una colección, normalmente de triángulos, que describe la superficie de un objeto
- Si la superficie que se representa tiene carácter curvo, entonces la representación mediante malla de polígonos es una aproximación, normalmente mejor cuanto mayor número de polígonos contenga



- Las caras de los polígonos se definen de tal manera que la normal apunte hacia el exterior del objeto:
  - para algoritmos de iluminación
  - para algoritmos de ocultamiento de superficies
  - plano que contiene el triángulo: Ax + By + Cz + D = 0
    - normal: (A, B, C)
  - vértices dados en sentido contrario a las agujas del reloj:
    - $\mathbf{v_0}$ - $\mathbf{v_1}$ - $\mathbf{v_2}$: normal apunta hacia exterior
    - normal según producto vectorial: $(\mathbf{v_1} - \mathbf{v_0}) \times (\mathbf{v_2} - \mathbf{v_0})$



Transformaciones geométricas aplicadas a normales

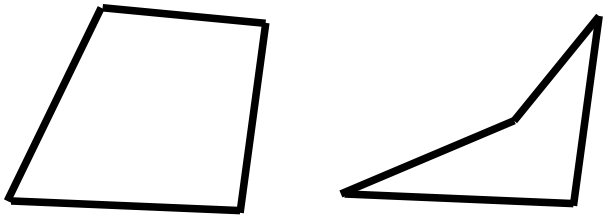
- Al aplicar una transformación geométrica a una malla que representa una superficie, también hay que aplicarla a las normales de sus triángulos.

- Si M es la transformación aplicada a un vértice **v** de la malla, y **n** es la normal en ese vértice, en general, M · **n** **no** devuelve la normal correctamente transformada

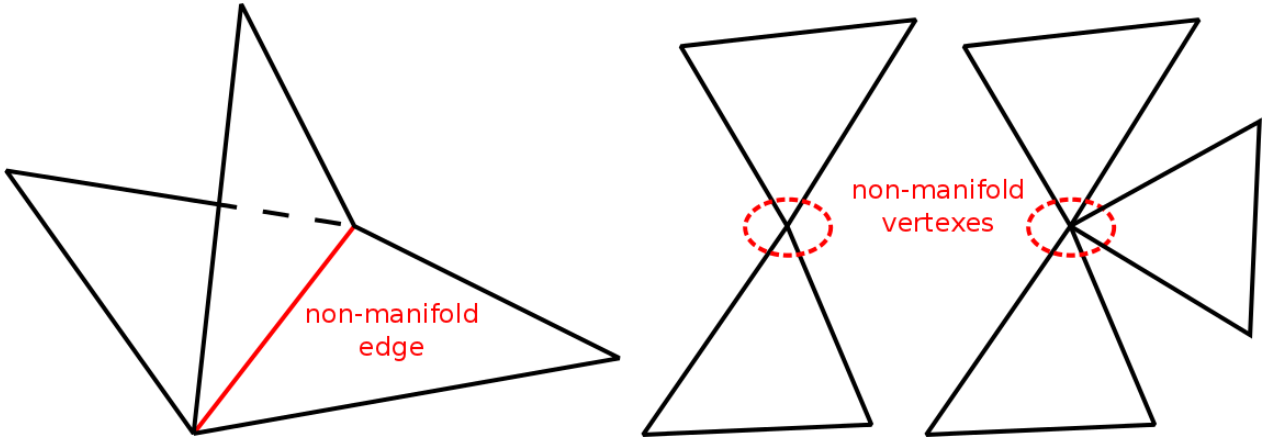
- Si **t** es la tangente a la superficie en **v**, entonces **t** y **n** son perpendiculares, **n** · t = 0, y después de aplicar la transformación al vértice, la nueva normal tiene que seguir siendo perpendicular a la tangente transformada. Como $M^{-1} \cdot M$ = Identidad, entonces:
  - $\mathbf{n}^T \cdot (M^{-1} \cdot M) \cdot \mathbf{t} = 0$
  - $(\mathbf{n}^T \cdot M^{-1}) \cdot (M \cdot \mathbf{t}) = 0$
  - como $(M \cdot \mathbf{t})$ es la tangente transformada, $(\mathbf{n}^T \cdot M^{-1})$ es la traspuesta de la normal transformada, por lo que
    - $(M^{-1})^T$ es la transformación a aplicar a la normal

3.3. Mallas de triángulos

- Clasificación de polígonos:
  - Convexos:
    - el ángulo formado por cada par de aristas consecutivas es menor de 180 grados
    - cualquier recta que une dos puntos del interior del polígono no corta ninguna de sus aristas
  - Cóncavos:
    - polígono que no es convexo

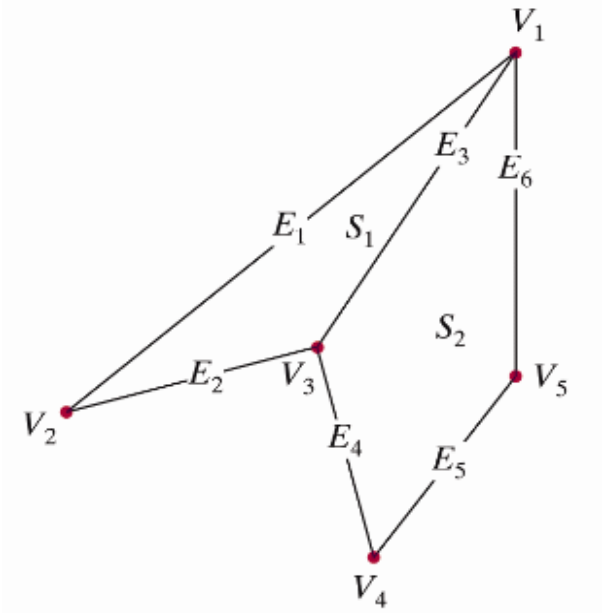


- Mallas de polígonos:
  - La mayoría de los modelos del mundo real están compuestos por mallas de polígonos que comparten sus vértices.
  - Las mallas de polígonos se utilizan para representar superficies, por lo que deben cumplir unos requisitos mínimos respecto a su topología: **variedad**:
    - Una malla es variedad si no tiene agujeros y divide el espacio en dos:



3.3. Mallas de triángulos

Representación de mallas de polígonos



- Ejemplo:
  - Hay que representar 5 vértices, 6 aristas y 2 polígonos
  - Cada vértice tiene unas coordenadas: $v_i = (x_i, y_i, z_i)$

- Representación simple:
  - listar cada polígono por su localización geométrica independiente

  - Ejemplo:
    - polígono 1: **(x_1, y_1, z_1)**, (x_2, y_2, z_2), **(x_3, y_3, z_3)**
    - polígono 2: **(x_1, y_1, z_1)**, **(x_3, y_3, z_3)**, (x_4, y_4, z_4), (x_5, y_5, z_5)

  - Problemas:
    - ineficiente y desestructurado
      - si se varía la posición de un vértice hay que cambiar las coordenadas en **todos** los polígonos en que aparece

Representación de mallas de polígonos

- Representación de geometría + topología:
  - geometría: localización de los vértices de la malla
  - topología: cómo se interconectan vértices, aristas y polígonos

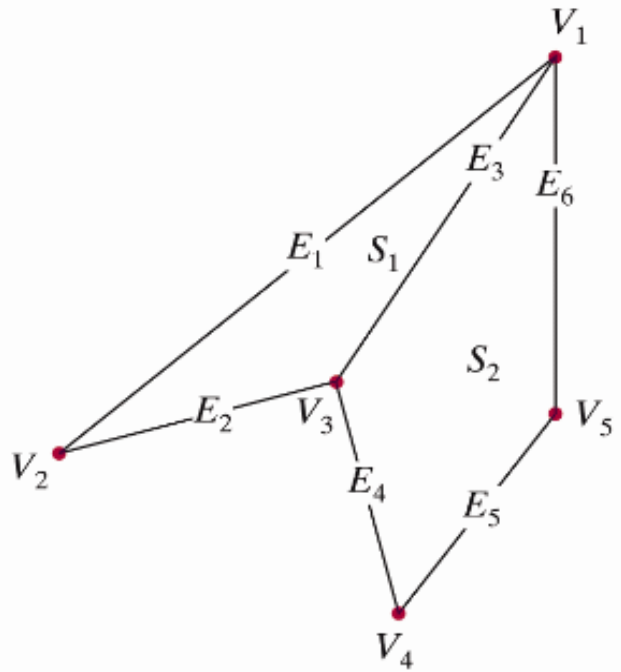
  - aunque la geometría cambie, la topología se mantiene

  - hay que utilizar estructuras de datos que mantengan la topología

VERTEX TABLE	
V_1:	x_1, y_1, z_1
V_2:	x_2, y_2, z_2
V_3:	x_3, y_3, z_3
V_4:	x_4, y_4, z_4
V_5:	x_5, y_5, z_5

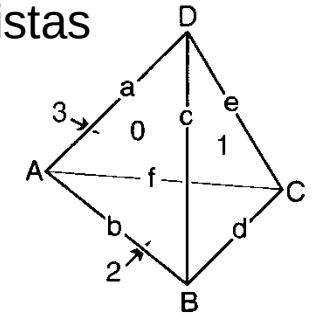
EDGE TABLE	
E_1:	V_1, V_2
E_2:	V_2, V_3
E_3:	V_3, V_1
E_4:	V_3, V_4
E_5:	V_4, V_5
E_6:	V_5, V_1

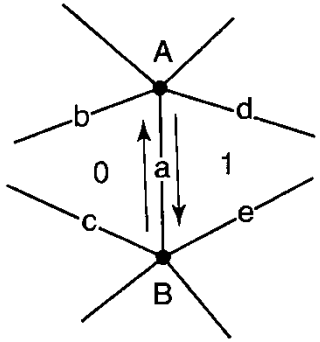
SURFACE-FACET TABLE	
S_1:	E_1, E_2, E_3
S_2:	E_3, E_4, E_5, E_6



Representación de mallas de polígonos

- Representación mediante Arista Alada (winged-edge):
  - guarda información topológica de adyacencia entre caras y aristas
  - el elemento principal es la arista, que almacena:
    - los vértices ordenados de la arista
    - las caras a las que pertenece la arista
    - las aristas anteriores y posteriores para cada cara
    - recorrido en **sentido anti-horario**





edge	vertex 1	vertex 2	face left	face right	pred left	succ left	pred right	succ right
a	B	A	0	1	c	b	d	e

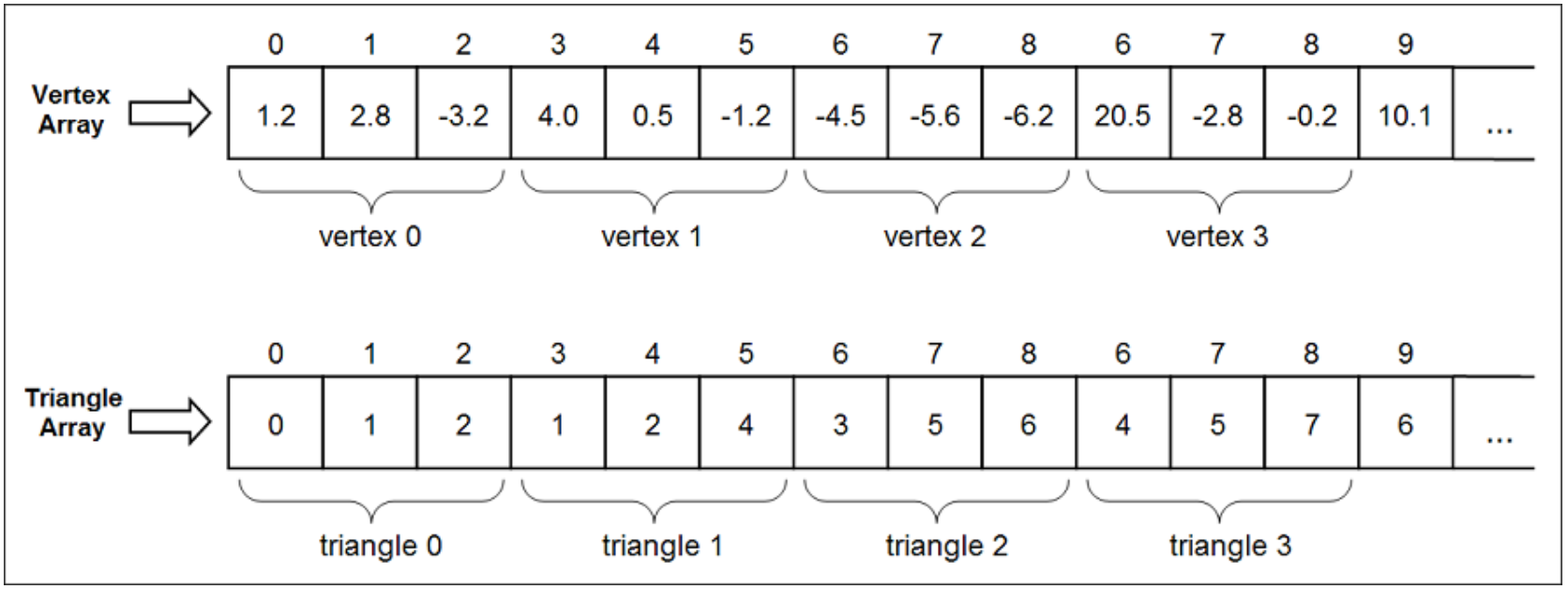
edge	vertex 1	vertex 2	face left	face right	pred left	succ left	pred right	succ right
a	A	D	3	0	f	e	c	b
b	A	B	0	2	a	c	d	f
c	B	D	0	1	b	a	e	d
d	B	C	1	2	c	e	f	b
e	C	D	1	3	d	c	a	f
f	C	A	3	2	e	a	b	d

vertex	edge
A	a
B	d
C	d
D	e

face	edge
0	a
1	c
2	d
3	a

Representación de mallas de polígonos

- Representación mediante mallas de triángulos indexadas:
  - es una manera muy habitual de representar mallas triangulares
  - se almacenan los vértices y los triángulos en arrays
    - un array de vértices
    - un array de triángulos que consiste en índices al array de vértices

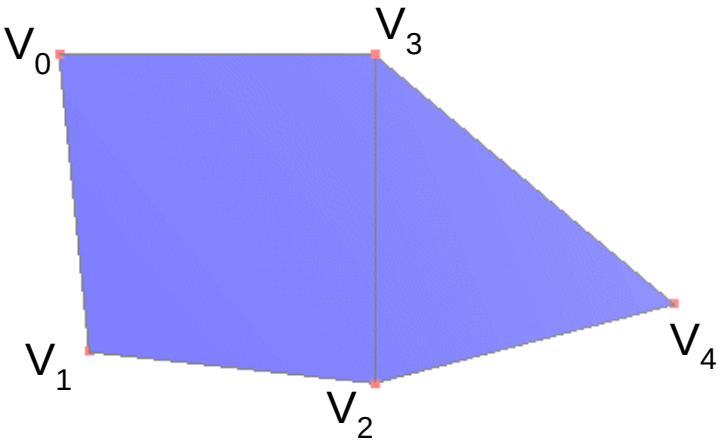


OpenGL: Visualización de polígonos

- Restricciones:
  - OpenGL sólo pinta de forma directa polígonos convexos (polígonos cóncavos tienen que ser descompuestos en convexos mediante **teseladores**)

- Representación básica:

```
glBegin(GL_POLYGON);
  glVertex3f(0,1,0);
  glVertex3f(0,0,0);
  glVertex3f(1,0,0);
  glVertex3f(1,1,0);
glEnd();
glBegin(GL_POLYGON);
  glVertex3f(1,1,0);
  glVertex3f(1,0,0);
  glVertex3f(2,0,-1);
glEnd();
```

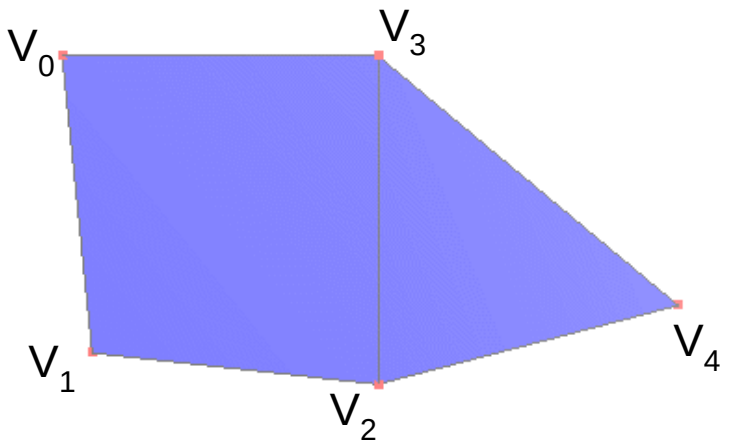


- Problema:
  - si se modifica la posición de un vértice, entonces **hay que modificar el código** de todos aquellos polígonos que compartan ese vértice

OpenGL: Array de vértices

- Problema: necesarias muchas llamadas OpenGL para la visualización de la malla:
  - En el peor de los casos visualizar la malla requiere:
    - 2 `glBegin()` y 2 `glEnd()`
    - 7 `glVertex()`
    - más funciones asociadas al color, iluminación, ...
  - muchas de estas funciones son redundantes y se pueden evitar
  - para mallas complejas pueden ser necesarias miles de llamadas a funciones OpenGL: consumo de muchos recursos y ralentización de la aplicación

- Solución: **array de vértices** de OpenGL (“OpenGL vertex array”)



OpenGL: Array de vértices

- Los arrays de vértices permiten reducir el número de llamadas necesarias para procesar las coordenadas de las mallas:
  - permiten almacenar las posiciones de la malla en la máquina de estados de OpenGL
- Los arrays de vértices también se pueden utilizar para manejar otros tipos de datos como los colores, normales y texturas
- Pasos para utilizar los arrays de vértices:
  1. Activar la utilización de arrays de vértices

```
glEnableClientState(GL_VERTEX_ARRAY);
```

  2. Especificar la localización y el formato del almacenamiento de los vértices

```
glVertexPointer(num_coord, tipo_coord, desp, array_coord);
```

     - `num_coord`: número de coordenadas para cada vértice
     - `tipo_coord`: tipo de las coordenadas (`GL_INT`, `GL_FLOAT`, ...)
     - `desp`: desplazamiento entre vértices consecutivos, sirve para tener en el mismo array información geométrica, de color, etc. Si sólo hay vértices debe valer 0
     - `aray_coord`: array con las coordenadas de los vértices

OpenGL: Array de vértices

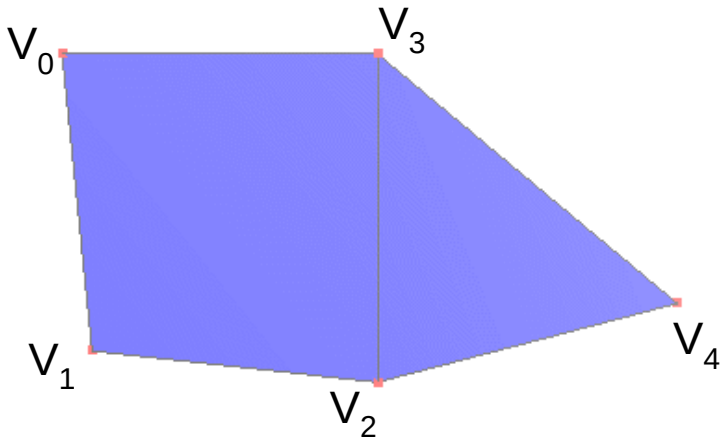
- Pasos para utilizar los arrays de vértices:
  3. Visualizar el objeto utilizando el array de vértices

     `glDrawElements(primitiva, num_ind, formato_ind, array_ind);`
     - `primitiva`: tipo de primitiva OpenGL a dibujar (`GL_POLYGON`, `GL_TRIANGLES`, `GL_QUADS`, ...)
     - `num_ind`: número de elementos en el array de índices
     - `formato_ind`: tipo de datos del array de índices (`GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT` o `GL_UNSIGNED_INT`)
     - `array_ind`: array de índices que definen cada elemento de la malla sobre el array de vértices pasado en `GlVertexPointer();`

     - si todos los polígonos tienen el mismo número de vértices:
       - `primitiva`: `GL_TRIANGLES` o `GL_QUADS`
       - `num_id`: número total de índices en `array_ind`
       - sólo una llamada a `glDrawElements();`
     - si en la malla hay polígonos con diferentes número de vértices:
       - son necesarias varias llamadas a `glDrawElements();`
       - `array_ind` debe apuntar a la posición concreta del array de índices para cada polígono
  4. Desactivar el array de vértices: `glDisableClientState(GL_VERTEX_ARRAY);`

OpenGL: Array de vértices

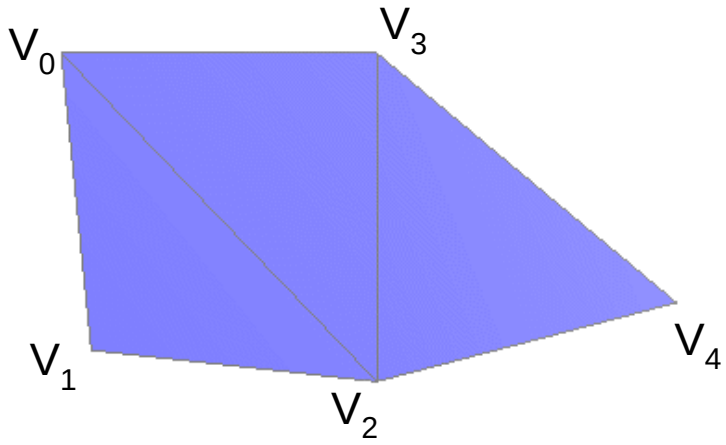
- Ejemplo: Malla **heterogénea** (NO es lo habitual)




```
GLfloat vertices[][3]={{0.0,1.0,0.0},
                       {0.0,0.0,0.0},
                       {1.0,0.0,0.0},
                       {1.0,1.0,0.0},
                       {2.0,0.0,-1.0}};
GLubyte indices[]={0,1,2,3,
                   3,2,4};

glEnableClientState(GL_VERTEX_ARRAY);

glVertexPointer(3,GL_FLOAT,0,vertices);

glDrawElements(GL_POLYGON,4,GL_UNSIGNED_BYTE,indices);
glDrawElements(GL_POLYGON,3,GL_UNSIGNED_BYTE,&indices[4]);

glDisableClientState(GL_VERTEX_ARRAY);
```

OpenGL: Array de vértices

- Ejemplo: Malla homogénea (normalmente **triángulos**)




```
GLfloat vertices[][3]={{0.0,1.0,0.0},
                       {0.0,0.0,0.0},
                       {1.0,0.0,0.0},
                       {1.0,1.0,0.0},
                       {2.0,0.0,-1.0}};
GLubyte indices[]={0,1,2,
                   0,2,3,
                   3,2,4};

glEnableClientState(GL_VERTEX_ARRAY);

glVertexPointer(3,GL_FLOAT,0,vertices);

glDrawElements(GL_TRIANGLES,9,GL_UNSIGNED_BYTE,indices);

glDisableClientState(GL_VERTEX_ARRAY);
```

OpenGL: Asignación de normales

- OpenGL permite asignar normales a:
  - un polígono completo
    - representar superficies planas
  - cada vértice de un polígono
    - sombreado de **Gouraud**
    - representar superficies curvas
- La normal da información sobre la orientación de la superficie en cada punto
- Útil para obtener la iluminación de los objetos y el ocultamiento de superficies
- La definición de normales se realiza con:
  `glNormal3{bsifd}[v](coordenadas_vector_normal);`
- Una vez establecido un valor de normal, este mismo valor se aplica a todas las llamadas a `glVertex()` posteriores

OpenGL: Asignación de normales

- Asignación de normal a:
  - un polígono completo:

```
glNormal3fv(vector_normal);
glBegin(GL_TRIANGLES);
  glVertex3fv(vertice0);
  glVertex3fv(vertice1);
  glVertex3fv(vertice2);
glEnd();
```

  - cada vértice de un polígono:

```
glBegin(GL_TRIANGLES);
  glNormal3fv(vector_normal0);
  glVertex3fv(vertice0);
  glNormal3fv(vector_normal1);
  glVertex3fv(vertice1);
  glNormal3fv(vector_normal2);
  glVertex3fv(vertice2);
glEnd();
```

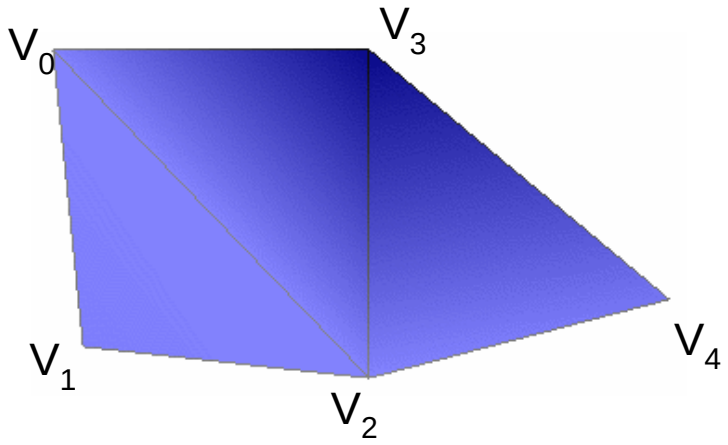
OpenGL: Asignación de normales

- Los cálculos con las normales se simplifican si el vector tiene módulo 1
  - `glEnable(GL_NORMALIZE);`
    - OpenGL automáticamente normaliza todos los vectores normales
    - también renormaliza los vectores después de aplicar escalados a los objetos

- Se puede asociar un **array de normales** a los polígonos generados con un array de vértices:
  - `glEnableClientState(GL_NORMAL_ARRAY);`
  - `glNormalPointer(tipo_coord, desp, array_normales);`
    - `tipo_coord`: tipo de las coordenadas (`GL_INT`, `GL_FLOAT`, ...)
    - `desp`: desplazamiento entre normales consecutivas. Si sólo hay normales debe valer 0
    - `aray_normales`: array con las coordenadas de las normales

OpenGL: Asignación de normales

- Ejemplo: Malla de triángulos



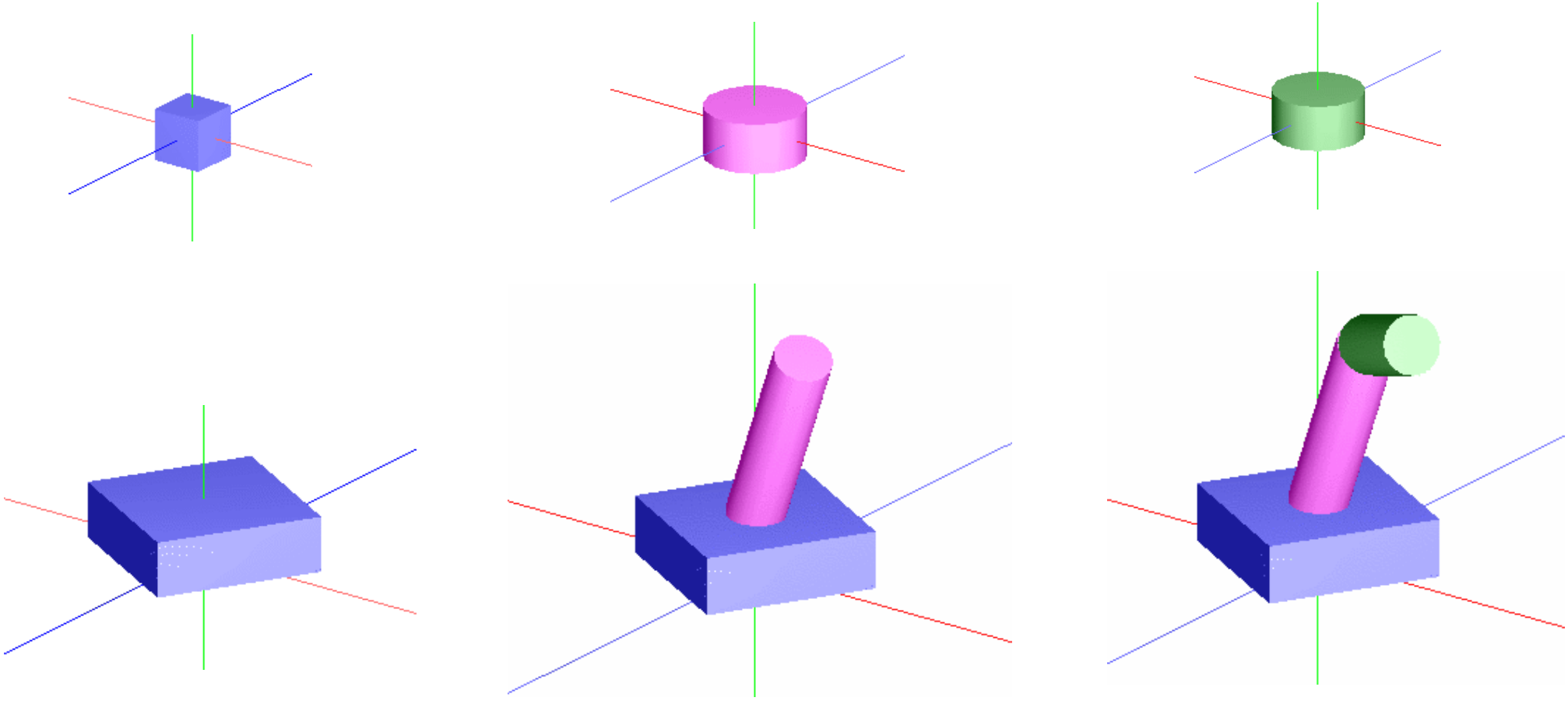

```
GLfloat vertices[][3]={{0.0,1.0,0.0},
                       {0.0,0.0,0.0},
                       {1.0,0.0,0.0},
                       {1.0,1.0,0.0},
                       {2.0,0.0,-1.0}};
GLfloat normales[][3]={{0.0,0.0,1.0},
                       {0.0,0.0,1.0},
                       {0.0,0.0,1.0},
                       {0.0,0.0,-1.0},
                       {1.0,0.0,0.0}};
GLubyte indices[]={0,1,2, 0,2,3, 3,2,4};

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3,GL_FLOAT,0,vertices);

glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(GL_FLOAT,0,normales);

glDrawElements(GL_TRIANGLES,9,GL_UNSIGNED_BYTE,indices);
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);
```

OpenGL: Práctica 3.A. Mallas de triángulos:

- Ver apartado **3.8. OpenGL: Práctica 3**

3.5. Grafos de escena

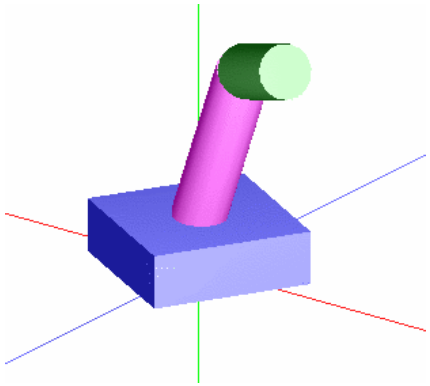
- Un problema fundamental en cualquier aplicación gráfica consiste en situar correctamente todos los objetos que constituyen la escena
- Para realizarlo se utilizan transformaciones, aunque las escenas y modelos complejos contienen muchas transformaciones que es necesario organizar



3.5. Grafos de escena

- La descripción de las escenas y los objetos complejos se pueden almacenar mediante una tabla de instanciación:
  - asignación de un símbolo a cada primitiva (letra, número, ...)
  - para cada símbolo se almacenan sus valores y las transformaciones de instanciación

- Ejemplo brazo articulado:
  - Primitiva cubo: símbolo 1
  - Primitiva cilindro: símbolo 2
  - Descripción del objeto final:



Símbolo	Parámetros	Transformaciones de Instanciación
1	ancho=3, alto=1, profundo=3	rotacion(-40,0,1,0), traslacion(0,0.5,0)
2	radio=0.5, altura=4	rotacion(-30,0,0,1), traslacion(0,2**+0.5**,0) **rotacion(-40,0,1,0)**
2	radio=0.5, altura=2	rotacion(-90,0,0,1), rotacion(-40**-30**,0,0,1), traslacion(0,1**+4+0.5**,0) **rotacion(-40,0,1,0)**

3.5. Grafos de escena

- Esta descripción de la escena presenta **importantes limitaciones**:
  - La tabla de instanciación no guarda la relaciones entre las diferentes partes del objeto final (el brazo superior está ligado al brazo inferior y el brazo inferior está ligado a la base)

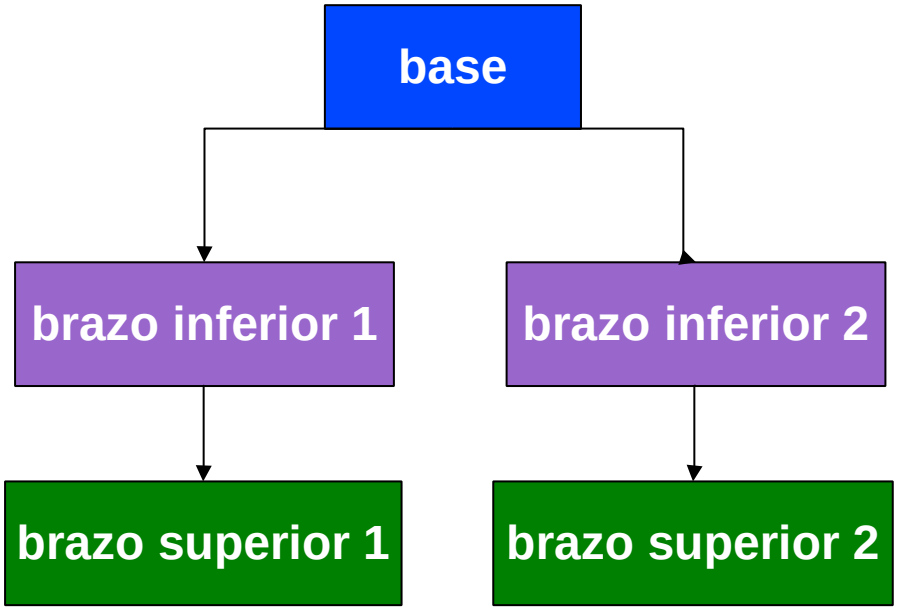
  - La tabla supone un modelo lineal de la escena u objeto
    - Ejemplo brazo articulado:
      - colocar base, colocar brazo inferior y colocar brazo superior
      - lo que existe verdaderamente es un modelo jerárquico:
        - base -> brazo inferior -> brazo superior
        - colocar base
        - en función de la posición actual de la base colocar el brazo inferior
        - en función de las posiciones actuales de la base y del brazo inferior colocar el brazo superior
      - la modificación de la posición de la base o del brazo inferior supondrá cambiar las matrices de instanciación del resto de elementos

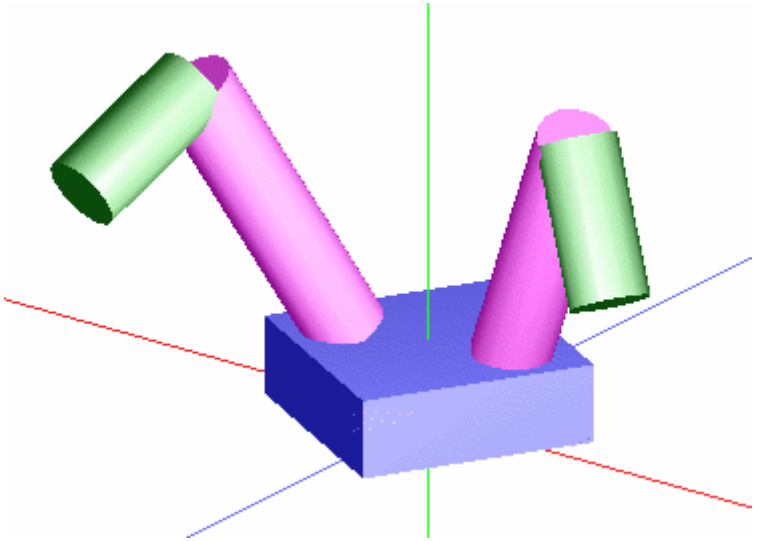
- Solución: es necesario utilizar otro tipo de estructuras para la representación de la escena

3.5. Grafos de escena

- La mayoría de las escenas y modelos complejos se pueden representar mediante una representación jerárquica, que se denomina **grafo de escena**

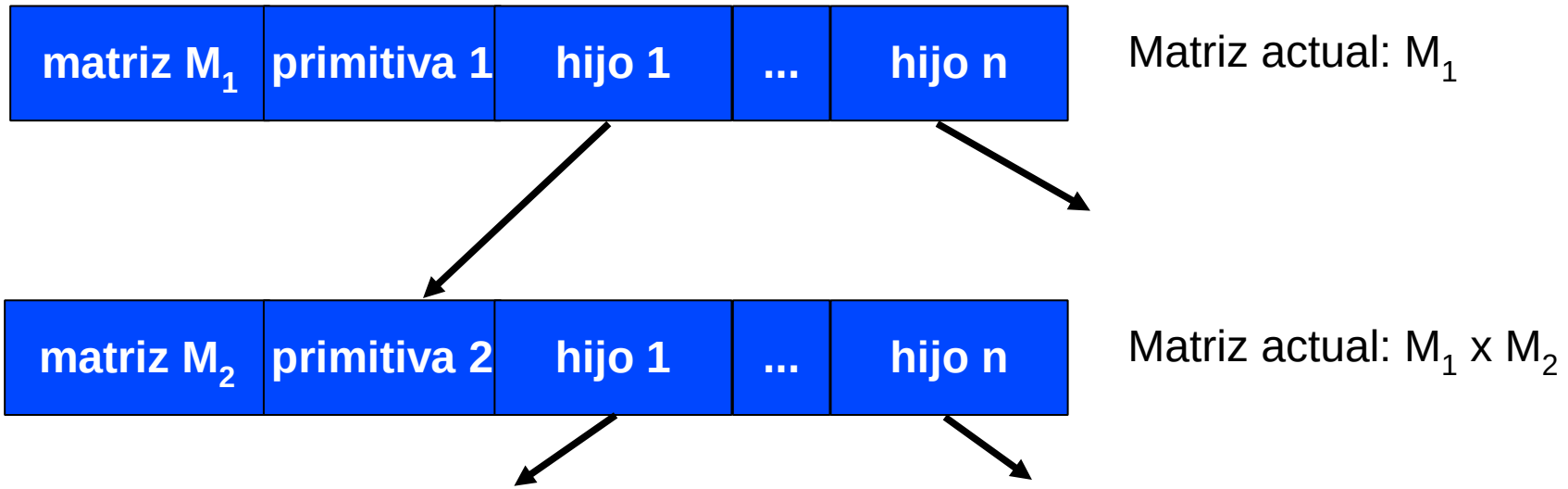
- Representación mediante **Árbol**:
  - La escena se genera recorriendo el árbol en **preorden**:
    - primero el padre y luego los hijos de izquierda a derecha
    - se implementa fácilmente con una pila
    - algoritmo independiente del modelo



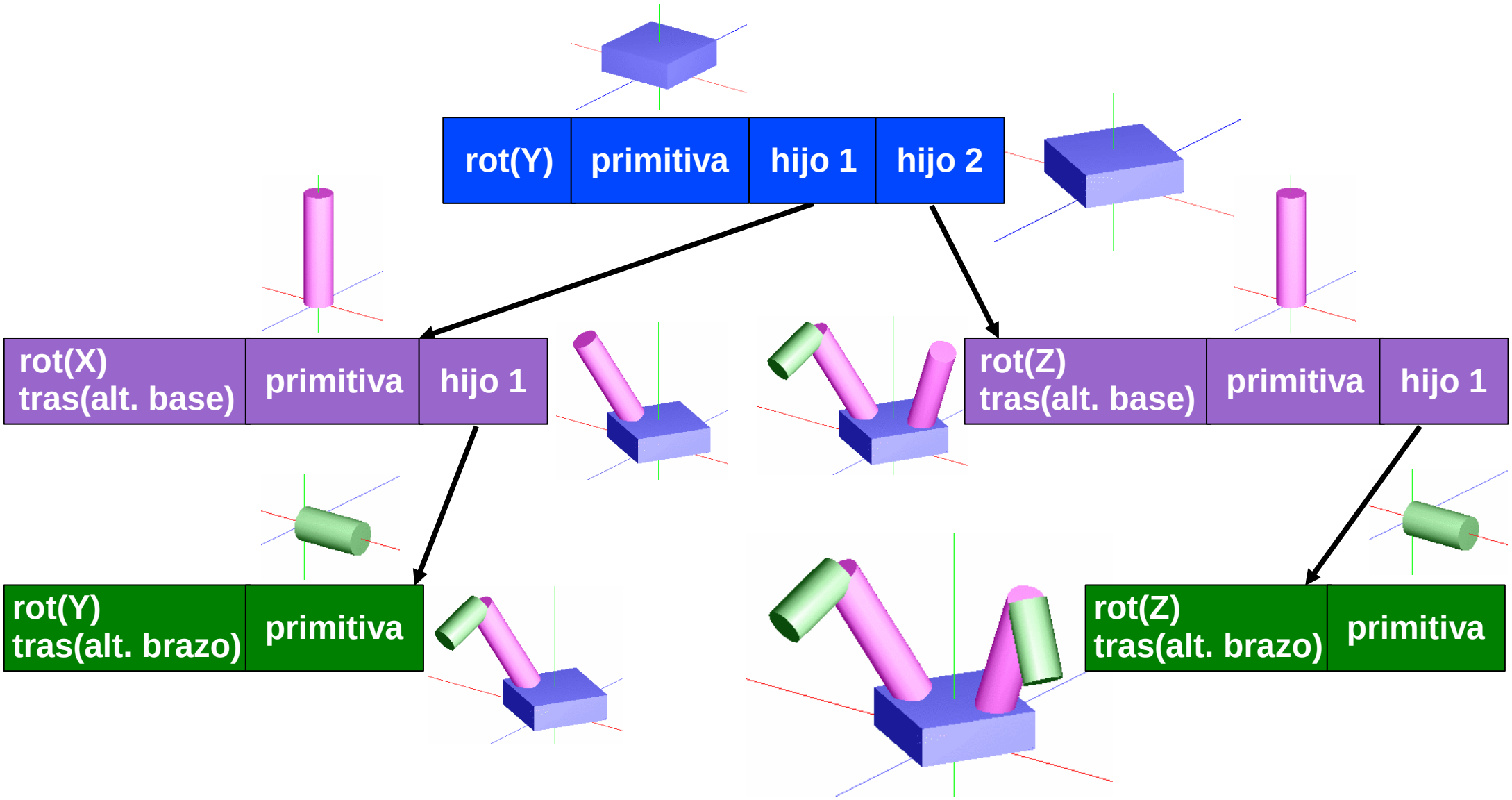


3.5. Grafos de escena

- Información a almacenar en cada nodo:
  1. matriz de transformación **local**
     - transformaciones locales del nodo y aquellas necesarias para situarlo respecto de la posición de la primitiva del padre, **NO** para situarlo en la posición final
     - al hacer el recorrido del árbol se compone con las transformaciones heredadas del padre
     - Ejemplo:
       - cada brazo (nodo hijo) se sitúa respecto a la primitiva de la base (nodo padre), si se mueve la base los brazos se mueven solidariamente
  2. primitiva a dibujar
  3. punteros a los nodos hijo

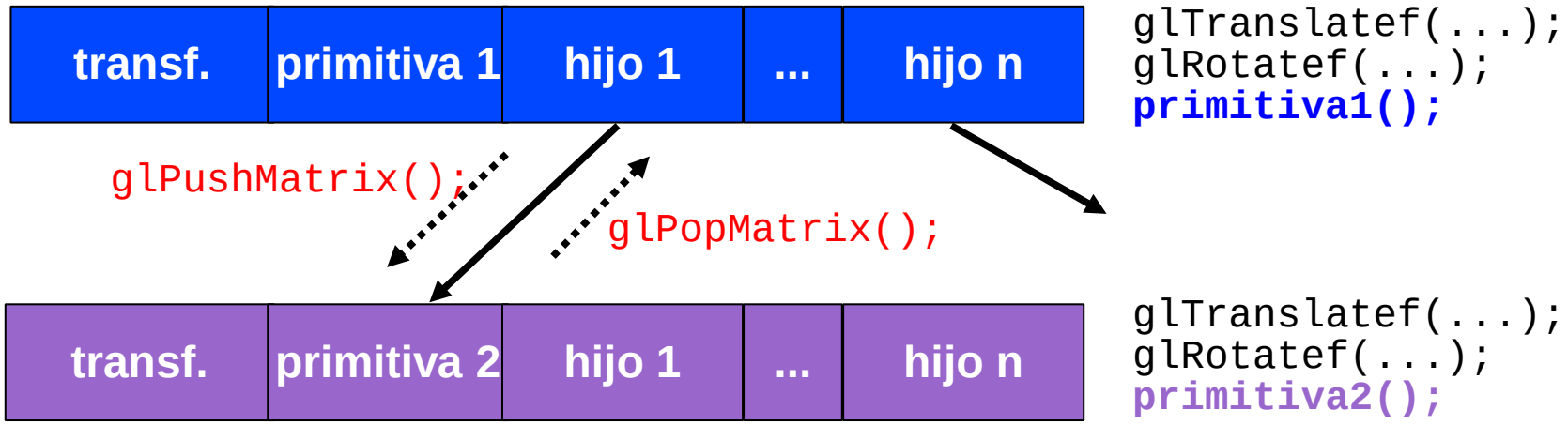


3.5. Grafos de escena



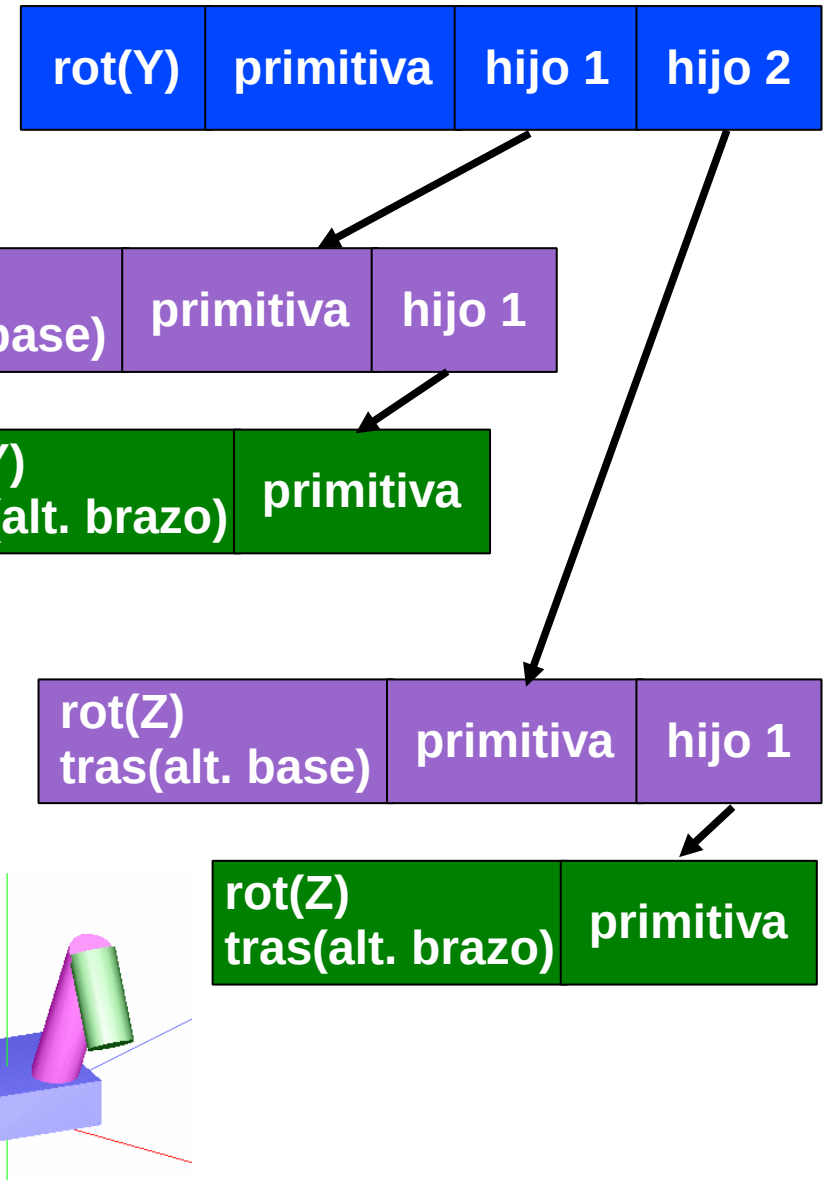
3.6. OpenGL: Grafos de escena

- Implementación a partir del árbol del modelo:
  - Empezar por la raíz y atravesar el árbol en preorden
  - Cuando se visita un nodo:
    - aplicar las transformaciones del nodo
    - dibujar la primitiva
  - Cuando se desciende de un nodo con algún hijo todavía sin visitar:
    - guardar la matriz actual de modelado: `glPushMatrix();`
  - Cuando se vuelve a un nodo padre con algún hijo todavía sin visitar:
    - borrar las transformaciones realizadas: `glPopMatrix();`



3.6. OpenGL: Grafos de escena

```
glRotatef(ANG_BASE,0,1,0);
base();

glPushMatrix();
  glTranslatef(-0.75,ALTURA_BASE,0.75);
  glRotatef(ANG_BRAZO_INF1,1,0,0);
  brazo_inferior();

  glTranslatef(0,ALTURA_BRAZO_INFERIOR,0);
  glRotatef(ANG_BRAZO_SUP1,0,1,0);
  brazo_superior();
glPopMatrix();

glTranslatef(0.75,ALTURA_BASE,-0.75);
glRotatef(ANG_BRAZO_INF2,0,0,1);
brazo_inferior();

glTranslatef(0,ALTURA_BRAZO_INFERIOR,0);
glRotatef(ANG_BRAZO_SUP1,0,0,1);
brazo_superior();
```

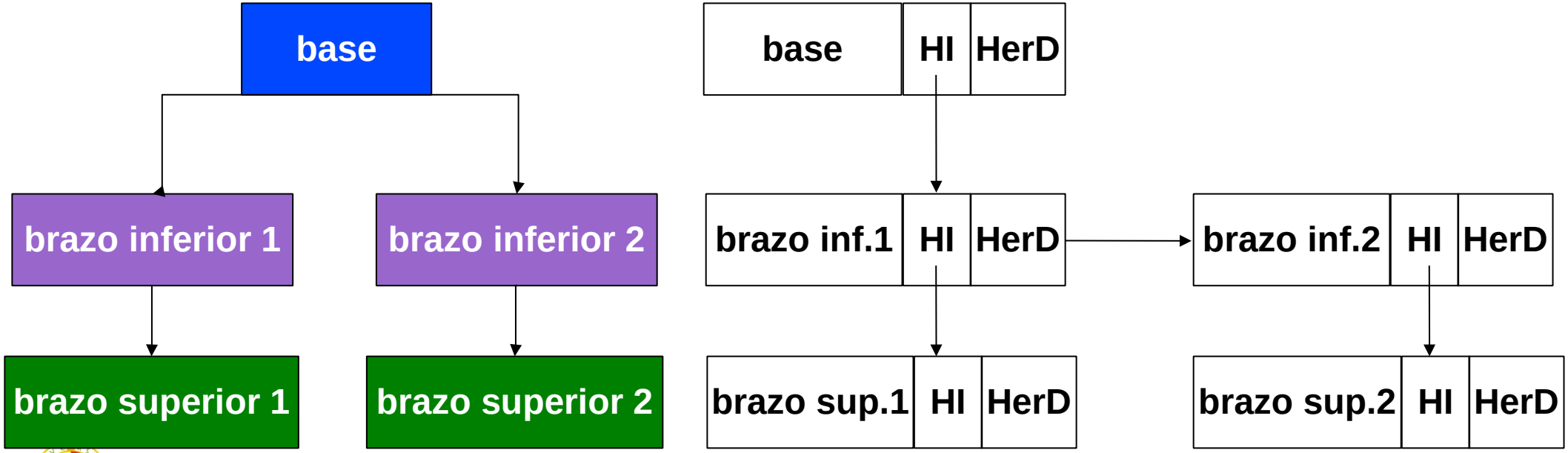


3.6. OpenGL: Grafos de escena

- Limitaciones de esta implementación:
  - el código refleja las relaciones entre las distintas partes de la escena
    - cambiar la escena implica cambiar el código
    - sólo se pueden modificar los parámetros de transformaciones y de las primitivas
  - código sólo sirve para una escena concreta
  - si se quieren añadir o eliminar objetos o partes del objeto hay que cambiar el código

Trabajo

- Para evitar que el código refleje la estructura del modelo:
  - Estructura de datos adicional para representar el árbol del modelo
  - Algoritmo independiente para realizar el recorrido del árbol
  - Posible implementación genérica de un árbol para almacenar los modelos:
    - como recorrido es en preorden: utilizar árbol “hijo izquierda-hermano derecha”
    - lista enlazada para hijo izquierda (HI)
    - lista enlazada para hermano derecha (HerD)



3.7. Selección e interacción con la escena

- Paradigma de aplicación gráfica interactiva:
  - Lo establece el sistema Sketchpad de Ivan Sutherland (1963):
    - En la pantalla se visualiza un objeto
    - El usuario selecciona (pick, “pincha”) el objeto con algún dispositivo de entrada (ratón, tableta digitalizadora, pulsación pantalla táctil, ...)
    - El objeto se modifica sobre la pantalla (se mueve, cambia forma, color, ...)
    - Repetir

- Dispositivos de entrada:
  - Dispositivos físicos:
    - se clasifican por sus propiedades físicas de funcionamiento
    - ejemplos: ratón, teclado, trackball, ...
  - Dispositivos lógicos:
    - se clasifican por el dato que devuelven a la aplicación que los utiliza
    - ejemplos: una coordenada de pantalla, un identificador de un objeto, ...

3.7.1. Dispositivos de entrada y gestión de eventos

Dispositivos de entrada

Dispositivos físicos:

- Absolutos:
  - devuelven a la aplicación una posición absoluta
  - ejemplo: tableta digitalizadora

- Relativos:
  - devuelven a la aplicación incrementos
  - en función de los incrementos se establece la posición absoluta
  - pueden tener sensibilidad variable
  - ejemplo: ratón, trackball, ...

3.7.1. Dispositivos de entrada y gestión de eventos

Dispositivos de entrada

Dispositivos lógicos:

- lectura en C desde la entrada estándar: `scanf("%d",&v);`
  - desde el código no se sabe cuál es el dispositivo físico de entrada
  - puede ser el teclado, un fichero, la salida de otro programa, ...
  - este código proporciona una **entrada lógica** de un número entero, independientemente del dispositivo físico que se utilice

- Clasificación estándar de los dispositivos lógicos:
  - **localizador**: especifica una posición
    - típico: ratón, tableta, trackball, joystick
    - teclado: cursor
    - opciones de menú
  - multi-localizador: especifica un conjunto de posiciones
    - típico: tableta
    - localizadores
  - cadena: especifica una entrada de texto
    - típico: teclado
    - reconocimiento de caracteres con multi-localizadores

3.7.1. Dispositivos de entrada y gestión de eventos

Dispositivos de entrada

Dispositivos lógicos:

- Clasificación estándar de los dispositivos lógicos:
  - valuador: especifica un valor escalar
    - útil para dar valores a transformación, parámetros de vista o iluminación
    - típico: paneles de control con diales
    - teclado, joystick, trackball
    - panel de control representado en pantalla y utilizar un localizador
  - elección: selecciona una opción de un menú
    - típico: ratón, teclado
    - accesos rápidos con teclas de función

  - **selector**: selecciona un componente de la pantalla
    - objetivo: transformar o editar la componente seleccionada
    - típico: ratón, trackball, joystick
    - posicionar cursor sobre el objeto a seleccionar y pulsar botón o tecla
      - **devuelve posición de coordenadas de pantalla**
      - **a partir de esta posición se selecciona un objeto, una cara, arista**, ...

3.7.1. Dispositivos de entrada y gestión de eventos

Gestión de eventos

- Las librerías gráficas que utilizan dispositivos lógicos deben proporcionar una serie de funciones para seleccionar:
  - el modo de interacción entre los dispositivos y el programa (iniciado por el programa, por el dispositivo, ...)
  - el dispositivo físico que se corresponde con cada dispositivo lógico (por ejemplo, el ratón se va a utilizar como localizador)
  - el momento de entrada y el dispositivo concreto para un conjunto de datos (por ejemplo, valores escalares tomados de un sensor)
- Modos de entrada:
  - modo solicitud:
    - el programa solicita al dispositivo los datos y espera hasta obtenerlos
  - modo sampleado:
    - el programa y el dispositivo funcionan de forma independiente
    - los datos nuevos suministrados por el dispositivo remplazan datos todavía no utilizados
    - cuando el programa solicita nuevos datos, toma los actuales
  - **modo eventos**:
    - el programa y el dispositivo funcionan de forma independiente
    - los datos nuevos suministrados por el dispositivo se almacenan en una cola
    - cuando el programa solicita nuevos datos, toma los primeros de la cola

3.7.2. OpenGL: Gestión de eventos

- Cada evento lleva asociada una medida:
  - las medidas son los parámetros del evento: posición ratón, estado del botón, ...

- En OpenGL la gestión de eventos se realiza con la librería GLUT

- Tipos de eventos en GLUT:
  - Ventana: cambiar tamaño, iconizar, exponer
  - Ratón: pulsar o soltar algún botón
  - Tableta digitalizadora: pulsar o soltar algún botón
  - Spaceball: pulsar o soltar algún botón
  - Dial: girar el dial un número de grados
  - Movimiento del ratón: mover el ratón con algún o si ningún botón presionado
  - Movimiento de la tableta digitalizadora: mover el cursor
  - Movimiento del spaceball: movimiento en 3D del cursor
  - Teclado: pulsar una tecla

  - Ocio: cuando no hay ningún evento

3.7.2. OpenGL: Gestión de eventos

- Cada evento lleva asociada una función de gestión del evento (*callback function*)
  - cuando ocurre un evento, se ejecuta la función de gestión correspondiente

- El bucle de gestión de eventos:
  - después de declarar las funciones de gestión de eventos
    - `glutMainLoop();`
  - en cada vuelta del bucle:
    - mirar la cola de eventos
    - para cada evento de la cola se ejecuta la función de gestión correspondiente
    - si no se ha definido la función, el evento se ignora

- Funciones de gestión de eventos de visualización:
  - `glutDisplayFunc(función);`
    - al abrir la ventana y al exponer la ventana
  - `glutPostRedisplay();`
    - fuerza la actualización de la ventana
  - `glutReshapeFunc(función);`
    - al redimensionar la ventana

3.7.2. OpenGL: Gestión de eventos

- Paso de valores del programa a las funciones de gestión de eventos:
  - los parámetros de estas funciones vienen fijados por GLUT
  - hay que utilizar variables globales

  - ejemplo:

```
GLfloat x;

void visualiza_escena() {
  /* visualización dependiente de x */
}

void main() {
  ...
  glutDisplayFunc(visualiza_escena);
}
```

3.7.2. OpenGL: Gestión de eventos

RATÓN

- Asignación de las funciones de gestión de eventos del ratón:
  - `glutMouseFunc(funcion_gestion_boton);`
  - `glutMotionFunc(funcion_gestion_movimiento_pulsado);`
  - `glutPassiveMotionFunc(funcion_gestion_movimiento_no_pulsado);`

- Declaración de las funciones:
  - `funcion_gestion_boton(GLint boton,GLint estado,GLint x,GLint y)`
    - boton: 0 (izquierdo), 1 (central), 2 (derecho), 3 (rueda adelante), 4 (rueda atrás)
    - estado: `GLUT_UP, GLUT_DOWN`. Para la rueda: devuelve los dos eventos
    - posición del ratón:
      - `(x,y)` en píxeles a partir de la esquina superior izquierda
      - **ATENCIÓN**: ventana visión OpenGL mide a partir de esquina **inferior** izquierda
        - correspondencia: OpenGL Y = altura_ventana - GLUT Y
      - **ATENCIÓN**: dimensiones de la ventana visión OpenGL no tienen que corresponder con dimensiones ventana GLUT
        - ejemplo: ventana GLUT: 400x300, ventana visión OpenGL 1x1
        - hay que **escalar** las posiciones GLUT para hacer corresponder con la ventana de visión

3.7.2. OpenGL: Gestión de eventos

RATÓN

- `funcion_gestion_movimiento_pulsado(GLint x, GLint y)`
  - `(x,y)` posición a la que se ha movido el ratón con algún botón pulsado

- `funcion_gestion_movimiento_no_pulsado(GLint x, GLint y)`
  - `(x,y)` posición a la que se ha movido el ratón con botones sin pulsar

3.7.2. OpenGL: Gestión de eventos

TECLADO

- Asignación de las funciones de gestión de eventos del teclado:
  - `glutKeyboardFunc(funcion_gestion_teclado);`
  - `glutKeyboardUpFunc(funcion_gestion_soltar_teclado);`
  - `glutSpecialFunc(funcion_gestion_teclado_especial);`

- Declaración de las funciones:
  - `funcion_gestion_(soltar_)teclado(GLubyte tecla, GLint x, GLint y)`
    - tecla: código ASCII de la tecla que se ha pulsado (soltado)
    - `(x,y)` posición del cursor cuando se pulsó la tecla (soltado)

  - `funcion_gestion_teclado_especial(GLubyte tecla, GLint x, GLint y)`
    - tecla: función: `GLUT_KEY_F1` a `GLUT_KEY_F12`, cursor: `GLUT_KEY_UP`, `GLUT_KEY_DOWN`, `GLUT_KEY_RIGHT`, `GLUT_KEY_LEFT`, página: `GLUT_PAGE_UP`, `GLUT_PAGE_DOWN`
    - `(x,y)` posición del cursor cuando se pulsó la tecla

- `int glutGetModifiers();`
  - devuelve: `GLUT_ACTIVE_SHIFT`, `GLUT_ACTIVE_CTRL`, `GLUT_ACTIVE_ALT`

3.7.2. OpenGL: Gestión de eventos

ANIMACIÓN

- Asignación de la función que se ejecuta **cuando no hay eventos**:
  - `glutIdleFunc(funcion_animacion);`

- Declaración de las funciones:
  - `funcion_animacion(void)`
    - contiene código que se ejecuta repetidamente si no hay eventos en el bucle de visualización de GLUT
    - se puede simular una animación si en esta función se incrementan/disminuyen los valores asociados a las posiciones de los objetos de la escena o de la cámara de visión
    - después de actualizar la descripción de la escena o de la cámara hay que revisualizar (`glutPostRedisplay()`)

3.7.3. Selección

- Objetivo de la selección interactiva:
  - situar el cursor sobre un objeto de la ventana, pulsar y seleccionarlo

- Problema:
  - varios objetos 3D se proyectan sobre el mismo punto 2D de la ventana

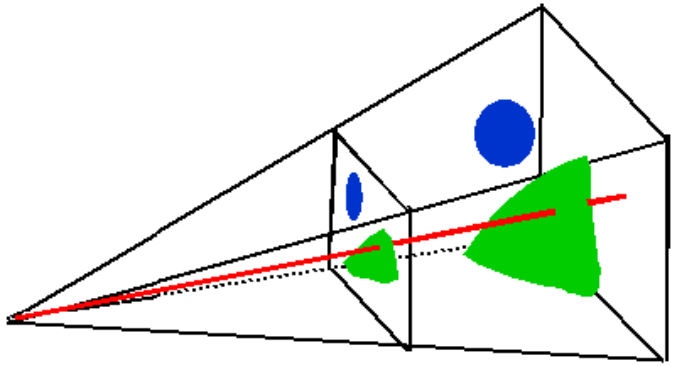
- Posibles soluciones:
  - intersección de rayos
  - cajas englobantes
  - buffer de color
  - **lista de impactos**

Trabajo

3.7.3. Selección

INTERSECCIÓN DE RAYOS

- Método:
  - seleccionar un punto de la ventana
  - lanzar un rayo desde el punto de visión pasando por el punto seleccionado
  - intersectar el rayo con todos los objetos de la escena
  - seleccionar el primer objeto intersectado por el rayo



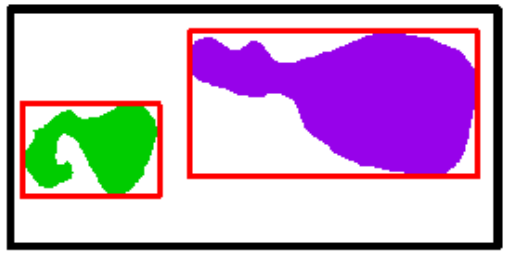
- Ventajas:
  - todo el procesamiento se realiza a nivel de la aplicación, no requiere librerías

- Inconvenientes:
  - difícil de programar
  - lento, ya que el método depende del número de objetos y de su complejidad

3.7.3. Selección

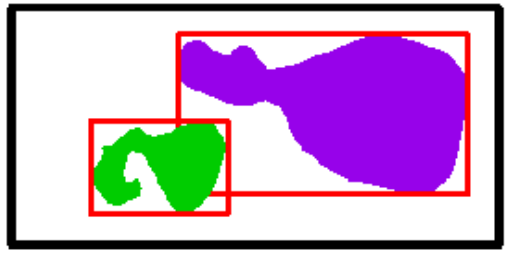
CAJAS ENGLOBANTES

- Método:
  - para cada objeto, calcular la caja englobante
  - cajas englobantes con caras alineadas a los ejes
  - la selección se realiza sobre las cajas



- Ventajas:
  - algoritmo muy simple
  - fácil mantener las cajas englobantes

- Inconvenientes:
  - precisión muy baja
  - solapamientos:
    - refinamiento de las cajas



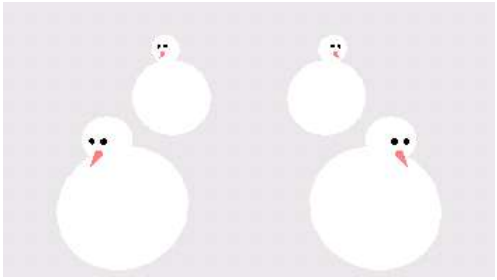
3.7.3. Selección

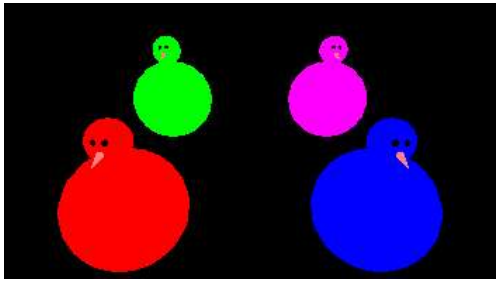
BUFFER DE COLOR

- Método:
  - utilizar un buffer adicional para la selección
    - cada objeto tiene asignado un color
  - seleccionar un punto de la ventana
  - redibujar los objetos en el buffer de color (sin sombreado y sin visualizar)
  - devolver objeto cuyo color sea el correspondiente a la posición seleccionada

- Ventajas:
  - algoritmo simple
  - alta precisión

- Inconvenientes:
  - se puede exceder el número de bits para el color
  - retraso al tener que redibujar en el buffer de color

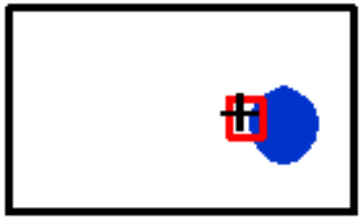


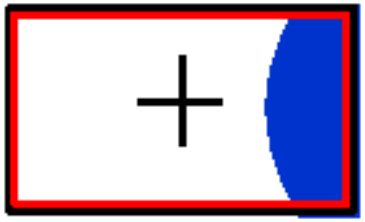


3.7.3. Selección

LISTA DE IMPACTOS

- Método:
  - asignar identificador a cada objeto
  - definir una región rectangular alrededor del cursor
  - recalcular volumen visión suponiendo que el viewport es la región definida
  - almacenar en una lista los objetos que impactan en la región
  - examinar la lista de impactos





- Ventajas:
  - método de selección más rápido:
    - soportado por OpenGL
    - algoritmo se basa en el recorte, no redibuja
  - flexibilidad en la precisión: en función del tamaño de la región

- Inconvenientes:
  - dificultad de implementar si no se utiliza una librería que soporte este método

3.7.4. OpenGL: Selección mediante lista de impactos

- Pasos para implementar la selección con lista de impactos:
  1. entrar en el **modo selección** de OpenGL
  2. redefinir volumen de visión con sólo un área pequeña alrededor del cursor
  3. revisualizar los objetos seleccionables
     - nombrar los objetos relevantes para la selección
  4. salir del modo selección
  5. identificar los objetos visualizados en el área alrededor del cursor
     - los nombres de estos objetos (impactos) los almacena OpenGL en un array (lista de impactos), se almacena:
       - el nombre del objeto
       - la **profundidad** mínima y máxima del objeto en el nuevo volumen de visión

3.7.4. OpenGL: Selección mediante lista de impactos

1. Entrar en modo selección de OpenGL:
   - Establecer dónde se van a almacenar los impactos:
     `glSelectBuffer(GLsizei tamaño,GLuint *buffer);`
     - `buffer`: array de `unsigned int` donde OpenGL almacenará los impactos
     - `tamaño`: el tamaño del array

   - Entrar en el modo selección de OpenGL:
     `glRenderMode(GL_SELECT);`
     - produce error si no se ha ejecutado antes `glSelectBuffer();`

2. Redefinir el volumen de visión alrededor de la posición de la ventana seleccionada:
   - guardar la matriz actual de proyección para poder restaurarla posteriormente:

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
```

3.7.4. OpenGL: Selección mediante lista de impactos

2. Redefinir el volumen de visión alrededor de la posición de la ventana seleccionada (continuación):
   - obtener el tamaño actual de la ventana:
     ```
     glGetIntegerv(GL_VIEWPORT,viewport);
     ```
   - generar el volumen de visión para la selección:
     ```
     gluPickMatrix(x,viewport[3]-y,ancho,alto,viewport);
     ```
     - `(x,y)` es la posición seleccionada (centro del área de selección)
     - `ancho` y `alto` definen el área de selección
   - calcular proyección:
     `gluPerspective(...);` o `glFrustum(...);` o `glOrtho(...);`
   - cargar de nuevo la matriz de modelado y establecer la cámara:
     ```
     glMatrixMode(GL_MODELVIEW);
     glLoadIdentity();
     glLookAt(...);
     ```

3.7.4. OpenGL: Selección mediante lista de impactos

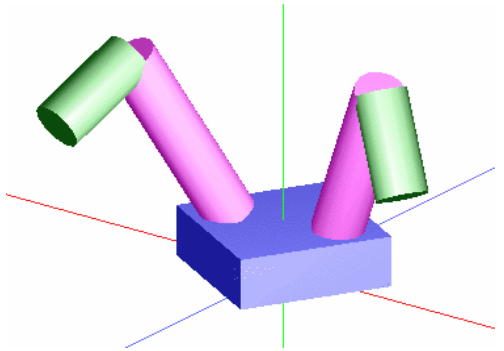
3. Revisualización de los objetos seleccionables:
   - Nombrar los objetos: La pila de nombres de OpenGL
     - los nombres (números enteros sin signo) se almacenan en una pila
     - inicialización de la pila de nombres:
       `glInitNames();`
     - añadir nombres a la pila:
       `glPushName(GLuint nombre);`
       - añade el nombre en el tope de la pila (máximo 64 nombres apilados)
     - eliminar nombres de la pila:
       `glPopName();`
       - elimina el nombre situado en el tope de la pila
     - remplazar el tope de la pila:
       `glLoadName(GLuint nombre);`
       - remplaza el tope de la pila poniendo el nombre
       - equivale a `glPopName(); glPushName(nombre);`

3.7.4. OpenGL: Selección mediante lista de impactos

3. Revisualización de los objetos seleccionables (continuación):
   - Nombrar los objetos: La pila de nombres de OpenGL
     - un mismo objeto puede tener varios nombres:
       - varias ejecuciones consecutivas de `glPushName`
       - útil para modelado jerárquico
     - estas funciones se ignoran cuando no se está en modo selección de OpenGL
     - estas funciones no se pueden utilizar dentro de `glBegin(); glEnd();`

   - Ejemplo de nombramiento de objetos (brazo articulado doble):

```
glPushName(BRAZO_1); // brazo 1 COMPLETO (1)
  glPushMatrix();
    transformaciones colocación brazo inferior 1;
    glPushName(BRAZO_INFERIOR); // brazo inferior 1 (3)
      brazo_inferior();
    glPopName();

    transformaciones colocación brazo superior 1;
    glPushName(BRAZO_SUPERIOR); // brazo superior 1 (4)
      brazo_superior();
    glPopName();
  glPopMatrix();
glPopName();
```



3.7.4. OpenGL: Selección mediante lista de impactos

4. Salir del modo selección:
   - restaurar la matriz de proyección inicial:
     ```
     glMatrixMode(GL_PROJECTION);
     glPopMatrix();
     ```
   - cargar de nuevo la matriz de modelado:
     ```
     glMatrixMode(GL_MODELVIEW);
     glFlush();
     ```
   - volver al modo de visualización normal para procesar los impactos:
     - `int glRenderMode(GL_RENDER);`
       - devuelve el número de impactos que se produjeron en el modo selección

3.7.4. OpenGL: Selección mediante lista de impactos

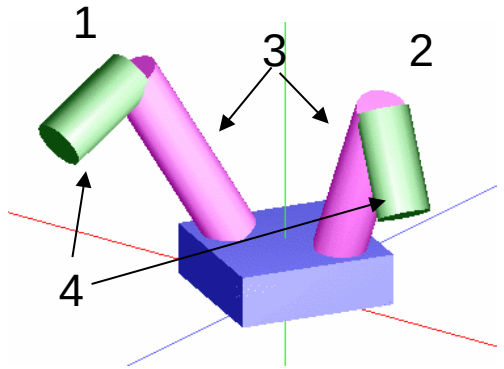
5. Identificación de objetos seleccionados (procesamiento de los impactos):
    - array de impactos:
        - es un array de **enteros sin signo**
        - almacena los impactos (objetos seleccionados) en el orden en que se dibujaron (tanto los que tienen nombre como los que no lo tienen)
        - objetos que podrían no verse en el modo de visualización normal al estar ocultos, pueden producir impactos

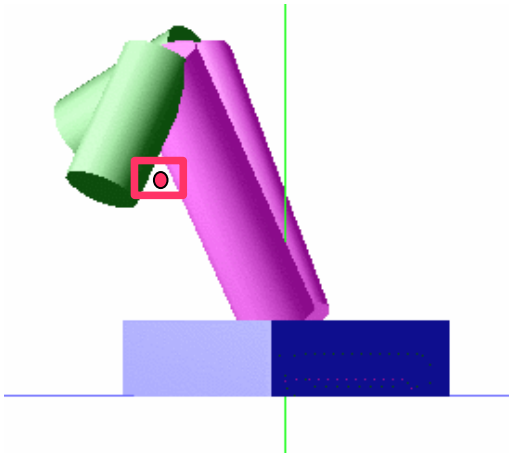
        - registros del array:
            - el tamaño de los registros es variable, en función del número de nombres
            - primer campo: número de nombres del impacto (puede valer 0)
            - segundo y tercer campo: profundidad mínima y máxima del impacto
                - profundidad de los vértices obtenidos al recortar contra el volumen de visión
                - valores entre 0 y 2^{32}-1
            - siguientes campos:
                - la secuencia de nombres del impacto (un nombre en cada posición del array), es decir, el contenido de la pila de nombres cuando se visualizó el objeto
                - **CUIDADO**: pueden no existir si el objeto no tiene nombre

3.7.4. OpenGL: Selección mediante lista de impactos

5. Identificación de objetos seleccionados (procesamiento de los impactos) (continuación):
   - array de impactos: ejemplo brazo articulado doble

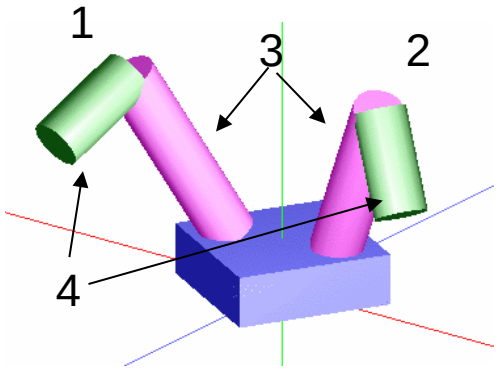
```
IMPACTO 0:
    numero nombres: 2
    minima Z: 147956111
    maxima Z: 154584207
    nombres: 2, 3
IMPACTO 1:
    numero nombres: 2
    minima Z: 129621519
    maxima Z: 137732879
    nombres: 2, 4
```

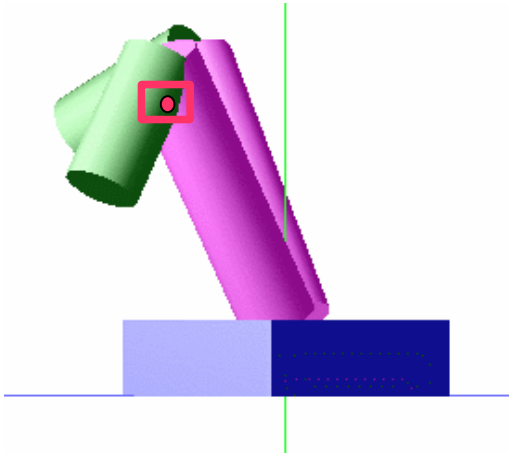




3.7.4. OpenGL: Selección mediante lista de impactos

5. Identificación de objetos seleccionados (procesamiento de los impactos):
   - array de impactos: ejemplo brazo articulado doble
     - IMPACTO 0:
       - numero nombres: 2
       - minima Z: 235295135
       - maxima Z: 251761823
       - nombres: 1, 3
     - IMPACTO 1:
       - numero nombres: 2
       - minima Z: 244231455
       - maxima Z: 263472671
       - nombres: 1, 4
     - IMPACTO 2:
       - numero nombres: 2
       - minima Z: 135245583
       - maxima Z: 158609807
       - nombres: 2, 3
     - IMPACTO 3:
       - numero nombres: 2
       - minima Z: **131503247**
       - maxima Z: 146950287
       - nombres: **2, 4**





3.7.4. OpenGL: Selección mediante lista de impactos

- Algunas consideraciones
  - utilizar una misma función de visualización de objetos tanto para modo selección y modo visualización:
    - las llamadas a funciones de pila de nombres se ignoran en modo visualización normal
    - más simple
    - más lento: en el modo selección se procesarán tanto los objetos seleccionables (los que tienen asignado un nombre) como los que no
  - utilizar una función especial de visualización de objetos para modo selección:
    - visualizar sólo los objetos seleccionables:
      - más eficiente
      - problema: objetos que no se visualicen en el modo selección no impiden que objetos detrás de ellos sean seleccionados (por ejemplo seleccionar objetos de diferentes habitaciones y no las paredes)
    - visualizar objetos seleccionables y aquellos que tapen partes de la escena

Práctica 3.A. Mallas de triángulos

- Ahora la clase `igvEscena3D` tiene un nuevo atributo `malla` de tipo puntero a `igvMallaTriangulos:`
  - malla de triángulos perteneciente a la escena que se visualiza en la ventana definida por un objeto de `igvInterfaz`

- clase **`igvMallaTriangulos`**: contiene funcionalidad **inicial** básica para representar y visualizar mallas de triángulos:

```
long int num_vertices; // número de vértices de la malla de triángulos
float *vertices; // array con las coordenadas de los vértices
float *normales; // array con las coordenadas de la normal a cada vértice

long int num_triangulos; // número de triángulos de la malla de triángulos
unsigned int *triangulos; // array con índices a vértices de cada triángulo
```

Práctica 3.B. Selección e Interacción en el Grafo de Escena

- clase `igvCamara`: nuevos atributos y métodos para calcular el volumen de visión asociado a la selección por lista de impactos:

```
modoCamara modo; // cámara para visualizar (IGV_VISUALIZACION) o para
                 // seleccionar (IGV_SELECCION)
int cursorX, cursorY; // posición del viewport seleccionada a partir de la
                      // cual hay que generar el volumen
                      // de visión para la selección por lista de impactos
int ancho_seleccion, alto_seleccion; // tamaño de la ventana para hacer la
                                     // selección por lista de impactos

// métodos para cambiar el modo de la cámara
void establecerSeleccion(int _ancho_seleccion, int _alto_seleccion,
                         int _cursorX, int _cursorY);
void establecerVisualizacion();

void aplicar(); // hay que implementar las transformaciones para selección
```

Práctica 3.B. Selección e Interacción en el Grafo de Escena

- clase `igvInterfaz`: nuevos atributos y métodos a implementar para realizar la selección de objetos sobre la ventana de visualización:

```
modoInterfaz modo; // IGV_VISUALIZAR: en la ventana se va a visualizar de
                   // manera normal la escena,
                   // IGV_SELECCIONAR: se ha pulsado sobre la ventana de
                   // visualización, la escena se debe visualizar en modo
                   // selección para el cálculo de la lista de impactos
int cursorX,cursorY; // pixel de la pantalla sobre el que está situado el
                     // ratón, por pulsar o arrastrar
int objeto_seleccionado; // código del objeto seleccionado, -1 si no hay
bool boton_retenido; // indica si botón pulsado (true) o soltado (false)
```

Práctica 3.B. Selección e Interacción en el Grafo de Escena

- clase `igvInterfaz`: nuevos atributos y **métodos a implementar** para realizar la selección de objetos sobre la ventana de visualización:

```
// métodos con funciones OpenGL para la selección mediante lista de impactos
inicia_seleccion(int TAMANO_LISTA_IMPACTOS, GLuint *lista_impactos);
finaliza_seleccion(int TAMANO_LISTA_IMPACTOS, GLuint *lista_impactos);

// función que procesa una lista con numero_impactos impactos y devuelve el
   objeto seleccionado
int procesar_impactos(int numero_impactos, GLuint *lista_impactos);

// métodos para el control de la pulsación y el arrastre del ratón
set_glutMouseFunc(GLint boton,GLint estado,GLint x,GLint y);
set_glutMotionFunc(GLint x,GLint y);
```

Biblografía

Computer Graphics with OpenGL, 4ª edición
Hearn, Baker y Carithers. Pearson, 2011.
Capítulos 4, 11, 13 y 20.

Fundamentals of Computer Graphics, 3ª edición
Shirley y Marschner. AK Peters, 2009
Capítulos 12 y 19.

OpenGL Programming Guide: the official guide to learning OpenGL, v. 3.0 and 3.1, 7ª edición
Addison-Wesley, 2010
Capítulos 2, 3, 13 y Apéndice A.