Open Inventor
3 - Primitive e Trasformazioni

Claudio Baldazzi *Maurizio Patrignani*

Università degli Studi di Roma Tre
Dip. di Informatica e Automazione
Via della Vasca Navale, 79 – 00146 Roma, Italy

25 marzo 1998

1 Introduzione

Questi appunti sono dedicati ai nodi di Open Inventor per la definizione delle primitive cosiddette lineari (sono cioè escluse le curve e le superfici, che saranno oggetto di una futura esercitazione) e per la definizione di trasformazioni.

Le informazioni contenute in questo documento e i listati discussi sono tratti, come sempre, dal libro "The Inventor Mentor" di Josie Wernecke edito dalla Addison Wesley.

2 Primitive

Alcune forme semplici che Open Inventor mette a disposizione del programmatore le abbiamo già incontrate. Queste sono le quattro primitive elementari:

- **Cubo**: `SoCube`, si può specificare la larghezza, l'altezza e la profondità.
- **Cono**: `SoCone`, si può specificare l'altezza ed il raggio della base.
- **Sfera**: `SoSphere`, si può specificare il raggio.
- **Cilindro**: `SoCylinder`, si può specificare l'altezza ed il raggio.

È evidente, però, che esse non sono sufficienti a descrivere scene complesse ed articolate, i cui oggetti generalmente sono difficilmente riconducibili ad una combinazione di oggetti così elementari.

Nel seguito vedremo alcune primitive più complesse, ed in particolare:

- **Face Set**: il nodo `SoFaceSet` permette di definire una serie di poligoni nello spazio (non necessariamente adiacenti), tramite le posizioni dei loro vertici.
- **Triangle Strip**: il nodo `SoTriangleStripSet` permette di definire una successione di triangoli, mediante una lista di vertici. Tre vertici consecutivi della lista costituiscono un triangolo. I triangoli sono così tutti adiacenti.

- **Quad Mesh**: il nodo `SoQuadMesh` permette di definire una serie di quadrilateri adiacenti, organizzati in linee e colonne, tramite le posizioni dei loro vertici.

Questi particolari nodi di Open Inventor non sono completamente definiti dai loro campi interni, ma necessitano di altri nodi, che potremmo chiamare "di appoggio", e che devono precedere i nodi stessi. Per esempio un nodo di tipo `SoFaceSet`, per la rappresentazione di un insieme di poligoni, abbisogna di un insieme di coordinate rappresentanti i loro vertici, ed un insieme di vettori rappresentanti la normale ad ogni poligono. Questi due insiemi sono definiti in due opportuni nodi che precedono il nodo `SoFaceSet`. Durante l'azione di rendering, i valori definiti dai due nodi di appoggio vengono memorizzati nello stato, ed il nodo `SoFaceSet` non fa altro che "raccogliere" questi valori ed interpretarli in maniera opportuna. Dal punto di vista tassonomico questi nodi "di appoggio" sono *property nodes*, di tipo *metrics*. Nel seguito noi useremo:

- SoCoordinate3: successione di punti definiti mediante le loro coordinate in tre dimensioni.
- SoNormal: successione di vettori definiti mediante le loro tre componenti lungo gli assi cartesiani (dimensionalmente sono terne di `float` come i precedenti).

Ecco un primo esempio di inizializzazione di un nodo `SoCoordinate3`:

```
SbVec3f vertici[6];
vertici[0] = SbVec3f(-3, -3, 0);
...
SoCoordinate3 *coordinate = new SoCoordinate3;
coordinate->point.setValues(0, 6, vertici);
```

Si noti che l'istruzione `coordinate->point.setValues(0, 6, vertici);` consente di inserire i valori a partire da una posizione specificata (in questo caso zero) nella successione di valori di nome `coordinate->point` del nodo `SoCoordinate3`.

Ecco un secondo esempio di inizializzazione di `SoCoordinate3`:

```
float vertici[6][3] = {
                          { 0, 30, 0}, {-2, 27, 2},
                          { 2, 27, 2}, { 0, 30, 0},
                          {-2, 27,-2}, {-2, 27, 2}
                      };
SoCoordinate3 *coordinate = new SoCoordinate3;
coordinate->point.setValues(0, 6, vertici);
```

L'inizializzazione di un nodo `SoNormal` è in tutto simile:

```
float normali[4][3] = {
                         {0.0, 1.0, 0.0}, {1.0,  0.0, 0.0},
                         {0.0, 0.0. 1.0}, {0.0, -1.0, 0.0}
                      };
SoNormal *myNormals = new SoNormal;
myNormals->vector.setValues(0, 4, normali);
```

2.1 La *Face Set*

Il `SoFaceSet` definisce un insieme di poligoni nello spazio, avvalendosi delle coordinate dei vertici e delle normali presenti nello stato (grazie ai nodi `SoCoordinate3` e `SoNormal` appena visti).

Ecco un esempio di descrizione un obelisco (privo della base):

```
/*--------------------------------------------------------------
 *  This is an example from the Inventor Mentor,
 *  chapter 5, example 1.
 *
 *  This example builds an obelisk using the Face Set node.
 *------------------------------------------------------------*/

//  Eight polygons. The first four are triangles
//  The second four are quadrilaterals for the sides.
static const float vertices[28][3] =
{
   { 0, 30, 0}, {-2,27, 2}, { 2,27, 2},            //front tri
   { 0, 30, 0}, {-2,27,-2}, {-2,27, 2},            //left  tri
   { 0, 30, 0}, { 2,27,-2}, {-2,27,-2},            //rear  tri
   { 0, 30, 0}, { 2,27, 2}, { 2,27,-2},            //right tri
   {-2, 27, 2}, {-4,0, 4}, { 4,0, 4}, { 2,27, 2},  //front quad
   {-2, 27,-2}, {-4,0,-4}, {-4,0, 4}, {-2,27, 2},  //left  quad
   { 2, 27,-2}, { 4,0,-4}, {-4,0,-4}, {-2,27,-2},  //rear  quad
   { 2, 27, 2}, { 4,0, 4}, { 4,0,-4}, { 2,27,-2}   //right quad
};

// Number of vertices in each polygon:
static long numvertices[8] = {3, 3, 3, 3, 4, 4, 4, 4};

// Normals for each polygon:
static const float norms[8][3] =
{
   {0, .555,  .832}, {-.832, .555, 0}, //front, left tris
   {0, .555, -.832}, { .832, .555, 0}, //rear, right tris

   {0, .0739,  .9973}, {-.9972, .0739, 0},//front, left quads
   {0, .0739, -.9973}, { .9972, .0739, 0},//rear, right quads
};

SoSeparator *
makeObeliskFaceSet()
{
   obelisk = new SoSeparator();
   obelisk->ref();

   // Define the normals used:
   SoNormal *myNormals = new SoNormal;
   myNormals->vector.setValues(0, 8, norms);
   obelisk->addChild(myNormals);
   SoNormalBinding *myNormalBinding = new SoNormalBinding;
   myNormalBinding->value = SoNormalBinding::PER_FACE;
   obelisk->addChild(myNormalBinding);

   // Define material for obelisk
   SoMaterial *myMaterial = new SoMaterial;
   myMaterial->diffuseColor.setValue(.4, .4, .4);
   obelisk->addChild(myMaterial);

   // Define coordinates for vertices
   SoCoordinate3 *myCoords = new SoCoordinate3;
```

```
49    myCoords->point.setValues(0, 28, vertices);
50    obelisk->addChild(myCoords);

51    // Define the FaceSet
52    SoFaceSet *myFaceSet = new SoFaceSet;
53    myFaceSet->numVertices.setValues(0, 8, numvertices);
54    obelisk->addChild(myFaceSet);

55    obelisk->unrefNoDelete();
56    return obelisk;
57 }
```

Note al codice:

linea 32 La procedura `makeObeliskFaceSet()` restituisce un `SoSeparator*`, cioè il puntatore al nodo radice dell'oggetto (un obelisco) creato. Il nodo `SoSeparator` viene allocato all'interno della funzione, che è dunque intesa per essere usata nel modo seguente:

`root->addChild(makeObeliskFaceSet());`

linea 35 Il nodo `obelisk`, di tipo `SoSeparator*`, è stato allocato dalla linea precedente, e sarà oggetto di future attenzioni (verranno definiti i suoi figli). Per impedire che venga cancellato da Open Inventor il programmatore si cautela istanziandolo forzosamente.

linea 53 L'istruzione `myFaceSet->numVertices.setValues(0, 8, numvertices);` configura il nodo di tipo `SoFaceSet*`, denunciando che si intende rappresentare un insieme di 8 facce, ognuna delle quali avrà il numero di vertici definiti in `numvertices` (che è un vettore di 8 elementi definito alla linea 21 come variabile globale).

Il primo valore passato (zero) è lo *starting index*, che segnala ad Open Inventor che gli otto valori che seguiranno vanno posti nella successione dei valori del campo `myFaceSet->numVertices` a partire dalla posizione zero fino alla posizione 7.

linea 55 In uscita dalla procedura `makeObeliskFaceSet()`, il contatore del nodo `obelisk` viene riportato a zero. L'istruzione `obelisk->unrefNoDelete();` ha l'effetto di decrementare il contatore relativo al nodo, senza che Open Inventor proceda alla cancellazione in caso di azzeramento del contatore stesso.

Open Inventor offre una versione più complessa del nodo `SoFaceSet`, il `SoIndexedFaceSet`, ia quale consente un risparmio di memoria rispetto al precedente, ed una programmazione più flessibile: un apposito campo del nodo `SoIndexedFaceSet` (il campo `coordIndex`) ha il ruolo di stabilire con quale ordine le coordinate presenti nello stato debbano essere considerate dal nodo. È evidente il vantaggio di poter utilizzare uno stesso punto più volte nel definire poligoni diversi che hanno quel punto in comune (eventualità relativamente frequente nella modellazione di superfici chiuse rappresentanti solidi).

2.2 La *Triangle Strip*

Il nodo `SoTriangleStripSet` costruisce una striscia di triangoli in base ai vertici descritti dalle attuali coordinate dello stato. È uno dei modi più veloci per disegnare oggetti poligonali in Open Inventor.

Nel seguente esempio una `SoTriangleStripSet` viene usata per definire una bandiera sostenuta da un pennone.

```
/*------------------------------------------------------------
 *  This is an example from the Inventor Mentor,
 *  chapter 5, example 3.
 *
 *  This example creates a TriangleStripSet. It creates
 *  a pennant-shaped flag.
 *-----------------------------------------------------------*/

// Positions of all of the vertices:

static const float vertexPositions[40][3] =
{
   {  0,   12,    0 }, {   0,   15,    0},
   {2.1, 12.1,  -.2 }, { 2.1, 14.6,  -.2},
   {  4, 12.5,  -.7 }, {   4, 14.5,  -.7},
   {4.5, 12.6,  -.8 }, { 4.5, 14.4,  -.8},
   {  5, 12.7,   -1 }, {   5, 14.4,   -1},
   {4.5, 12.8, -1.4 }, { 4.5, 14.6, -1.4},
   {  4, 12.9, -1.6 }, {   4, 14.8, -1.6},
   {3.3, 12.9, -1.8 }, { 3.3, 14.9, -1.8},
   {  3,   13, -2.0 }, {   3, 14.9, -2.0},
   {3.3, 13.1, -2.2 }, { 3.3, 15.0, -2.2},
   {  4, 13.2, -2.5 }, {   4, 15.0, -2.5},
   {  6, 13.5, -2.2 }, {   6, 14.8, -2.2},
   {  8, 13.4,   -2 }, {   8, 14.6,   -2},
   { 10, 13.7, -1.8 }, {  10, 14.4, -1.8},
   { 12,   14, -1.3 }, {  12, 14.5, -1.3},
   { 15, 14.9, -1.2 }, {  15,   15, -1.2},

   {-.5, 15,   0 }, { -.5, 0,   0},   // the flagpole
   {  0, 15,  .5 }, {   0, 0,  .5},
   {  0, 15, -.5 }, {   0, 0, -.5},
   {-.5, 15,   0 }, { -.5, 0,   0}
};

// Number of vertices in each strip.
static long numVertices[2] =
{
   32, // flag
   8   // pole
};

// Colors for the 12 faces
static const float colors[2][3] =
{
   { .5, .5,  1 }, // purple flag
   { .4, .4, .4 }, // grey flagpole
};

// Routine to create a scene graph representing a pennant.
SoSeparator *
makePennant()
{
   SoSeparator *result = new SoSeparator;
   result->ref();

   // A shape hints tells the ordering of polygons.
   // This insures double sided lighting.
   SoShapeHints *myHints = new SoShapeHints;
   myHints->vertexOrdering = SoShapeHints::COUNTERCLOCKWISE;
```

```
55    result->addChild(myHints);

56    // Define colors for the strips
57    SoMaterial *myMaterials = new SoMaterial;
58    myMaterials->diffuseColor.setValues(0, 2, colors);
59    result->addChild(myMaterials);
60    SoMaterialBinding *myMaterialBinding = new SoMaterialBinding;
61    myMaterialBinding->value = SoMaterialBinding::PER_PART;
62    result->addChild(myMaterialBinding);

63    // Define coordinates for vertices
64    SoCoordinate3 *myCoords = new SoCoordinate3;
65    myCoords->point.setValues(0, 40, vertexPositions);
66    result->addChild(myCoords);

67    // Define the TriangleStripSet, made of two strips.
68    SoTriangleStripSet *myStrips = new SoTriangleStripSet;
69    myStrips->numVertices.setValues(0, 2, numVertices);
70    result->addChild(myStrips);

71    result->unrefNoDelete();
72    return result;
73 }
```

Note al codice:

linea 47 La procedura `makePennant()` restituisce un `SoSeparator*`, ed è intesa per essere chiamata nel modo seguente: `root->addChild(makePennant());`

linea 50 Come negli esempi precedenti il contatore della radice del sottoalbero che stiamo per definire viene incrementato forzosamente per impedirne la cancellazione automatica.

linea 69 L'istruzione in questa linea configura il nodo `SoTriangleStripSet*`) in maniera che esso interpreti le coordinate che trova nello stato come due Triangle Strip, una composta da `numVertices[0]` punti e l'altra da `numVertices[1]` punti.

Il vettore `long numVertices[2]` è una variabile globale definita ed inizializzata nella linea 40 e seguenti.

linea 71 Il contatore del nodo `SoSeparator` viene decrementato senza che questa azione ne provochi la cancellazione.

Anche del nodo `SoTriangleStripSet` esiste una versione indicizzata, la `SoIndexedTriangleStripSet` analoga alla `SoIndexedFaceSet`.

2.3 Quad Mesh

Il nodo `SoQuadMesh` costruisce una successione di quadrilateri a partire dai vertici costituiti delle coordinate correnti dello stato.

Si consideri il seguente esempio nel quale un `SoQuadMesh` viene utilizzato per descrivere un oggetto tubolare piegato ad arco:

```
/*--------------------------------------------------------------
 *  This is an example from the Inventor Mentor,
 *  chapter 5, example 4.
 *
```

```
 5  *  This example creates the St. Louis Arch using a QuadMesh.
 6  *------------------------------------------------------------*/

 7 // Positions of all of the vertices:
 8 static const float vertexPositions[160][3] =
 9 {  // 1st row
10    {-13.0,  0.0, 1.5}, {-10.3, 13.7, 1.2}, { -7.6, 21.7, 1.0},
11    { -5.0, 26.1, 0.8}, { -2.3, 28.2, 0.6}, { -0.3, 28.8, 0.5},
12    {  0.3, 28.8, 0.5}, {  2.3, 28.2, 0.6}, {  5.0, 26.1, 0.8},
13    {  7.6, 21.7, 1.0}, { 10.3, 13.7, 1.2}, { 13.0,  0.0, 1.5},
14    // 2nd row
15    {-10.0,  0.0, 1.5}, { -7.9, 13.2, 1.2}, { -5.8, 20.8, 1.0},
16    { -3.8, 25.0, 0.8}, { -1.7, 27.1, 0.6}, { -0.2, 27.6, 0.5},
17    {  0.2, 27.6, 0.5}, {  1.7, 27.1, 0.6}, {  3.8, 25.0, 0.8},
18    {  5.8, 20.8, 1.0}, {  7.9, 13.2, 1.2}, { 10.0,  0.0, 1.5},
19    // 3rd row
20    {-10.0,  0.0,-1.5}, { -7.9, 13.2,-1.2}, { -5.8, 20.8,-1.0},
21    { -3.8, 25.0,-0.8}, { -1.7, 27.1,-0.6}, { -0.2, 27.6,-0.5},
22    {  0.2, 27.6,-0.5}, {  1.7, 27.1,-0.6}, {  3.8, 25.0,-0.8},
23    {  5.8, 20.8,-1.0}, {  7.9, 13.2,-1.2}, { 10.0,  0.0,-1.5},
24    // 4th row
25    {-13.0,  0.0,-1.5}, {-10.3, 13.7,-1.2}, { -7.6, 21.7,-1.0},
26    { -5.0, 26.1,-0.8}, { -2.3, 28.2,-0.6}, { -0.3, 28.8,-0.5},
27    {  0.3, 28.8,-0.5}, {  2.3, 28.2,-0.6}, {  5.0, 26.1,-0.8},
28    {  7.6, 21.7,-1.0}, { 10.3, 13.7,-1.2}, { 13.0,  0.0,-1.5},
29    // 5th row
30    {-13.0,  0.0, 1.5}, {-10.3, 13.7, 1.2}, { -7.6, 21.7, 1.0},
31    { -5.0, 26.1, 0.8}, { -2.3, 28.2, 0.6}, { -0.3, 28.8, 0.5},
32    {  0.3, 28.8, 0.5}, {  2.3, 28.2, 0.6}, {  5.0, 26.1, 0.8},
33    {  7.6, 21.7, 1.0}, { 10.3, 13.7, 1.2}, { 13.0,  0.0, 1.5}
34 };

35 // Routine to create a scene graph representing an arch.
36 SoSeparator *
37 makeArch()
38 {
39    SoSeparator *result = new SoSeparator;
40    result->ref();

41    // Define the material
42    SoMaterial *myMaterial = new SoMaterial;
43    myMaterial->diffuseColor.setValue(.78, .57, .11);
44    result->addChild(myMaterial);

45    // Define coordinates for vertices
46    SoCoordinate3 *myCoords = new SoCoordinate3;
47    myCoords->point.setValues(0, 60, vertexPositions);
48    result->addChild(myCoords);

49    // Define the QuadMesh.
50    SoQuadMesh *myQuadMesh = new SoQuadMesh;
51    myQuadMesh->verticesPerRow = 12;
52    myQuadMesh->verticesPerColumn = 5;
53    result->addChild(myQuadMesh);

54    result->unrefNoDelete();
55    return result;
56 }
```

Note al codice:

linea 37 Anche in questo caso la procedura `makeArch()` restituisce un `SoSeparator*`, ed è intesa per essere usata nel modo seguente:

```
root->addChild(makeArch());
```

linea 52 Viene dichiarato in questa linea e nella successiva come debbano essere interpretate le coordinate contenute nello stato: le terne di valori (una terna per punto) compongono una successione di cinque linee, ognuna delle quali è composta da dodici punti. Ogni quaterna di punti composta da due punti successivi su una linea e i punti omonimi sulla linea successiva, costituirà un quadrilatero. Si noti che le linee 10, 11, 12 e 13 sono identiche alle linee 30, 31, 32 e 33 rispettivamente: l'ultima riga coincide dunque con la prima. L'oggetto generato da questo `SoQuadMesh` è dunque una 'superficie' chiusa su sé stessa (una specie di tubo) composta da quadrilateri (ovviamente i due estremi rimarranno aperti).

3 Trasformazioni Geometriche

Come è stato già constatato nelle lezioni scorse, al contrario degli altri *property nodes*, i nodi che rappresentano le trasformazioni geometriche non sostituiscono la matrice corrente di trasformazione nel *traversal state*, bensì hanno un effetto cumulativo su di essa e, a causa di questo, assume estrema importanza l'ordine in cui tali trasformazioni vengono inserite nel grafo della scena. In figura 1 si

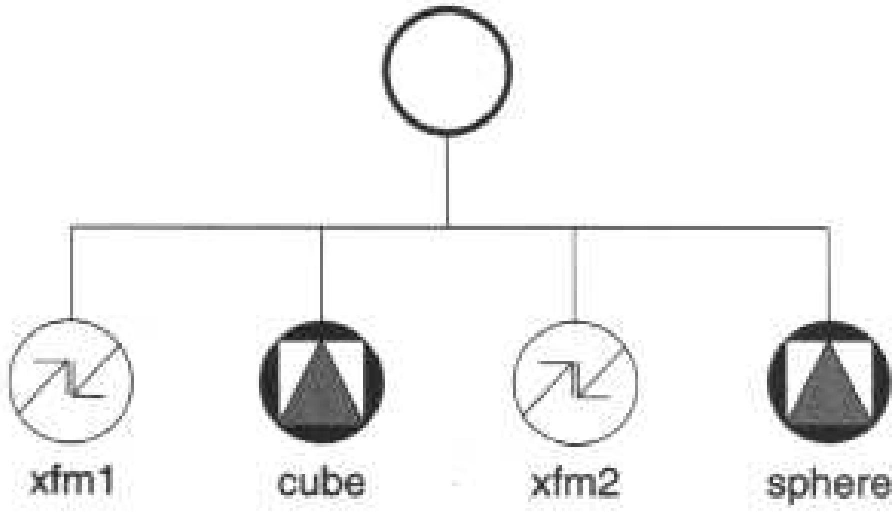

Figura 1: Effetto di concatenazione di trasformazioni geometriche

può vedere come il cubo sia affetto solo dalla trasformazione *xfm1* mentre la sfera dalla concatenazione di *xfm1* e *xfm2*.

Un nodo che effettua una trasformazione geometrica appartiene alla classe **SoTransform**, che contiene i seguenti campi per realizzare le trasformazioni:

- **translation** specifica la traslazione lungo gli assi x, y e z. Ad esempio il codice

```
SoTransform*  xform = new SoTransform;

xform->translation.setValue( 1.0, 2.0, -1.5 );
```

effettua una traslazione lungo gli assi x, y e z rispettivamente di 1.0, 2.0 e -1.5. Il valore di default alla creazione del nodo è pari alla tupla $[0.0, 0.0, 0.0]$, quindi nessuna traslazione.

- **rotation** specifica la rotazione intorno ad un asse qualunque, passante per l'origine del sistema di riferimento, di un certo numero di gradi espressi in radianti. Ad esempio

  ```
  xform->rotation.setValue( 1.0, 0.0, 0.0, 1.5 );
  ```

  significa effettuare una rotazione intorno all'asse x di 1.5 radianti. I primi tre numeri reali identificano un vettore applicato nell'origine che rappresenta il versore (non normalizzato) dell'asse di rotazione voluto. Se l'asse non deve passare per l'origine, va usato anche il metodo **center**. Il valore di default alla creazione del nodo è $[0.0, 0.0, 1.0, 0.0]$, quindi rotazione intorno l'asse z di 0 radianti, cioè nessuna rotazione.

- **scaleFactor** specifica uno scaling lungo le direzioni x, y e z. Ad esempio

  ```
  xform->scaleFactor.setValue( 1.0, 2.0, 0.75 );
  ```

  effettua uno scalamento lungo x, y e z rispettivamente di 1.0, 2.0 e 0.75. Il valore di default alla creazione del nodo è $[1.0, 1.0, 1.0]$, cioè nessuno scalamento.

- **scaleOrientation** specifica una rotazione dell'oggetto da effettuare subito prima dello scalamento. È come se, nello scalamento, si effettuasse per prima cosa una rotazione dell'oggetto, dopo lo scalamento vero e proprio, e dopo una rotazione contraria alla precedente per poter riposizionare l'oggetto al posto primitivo. La rotazione non è in realtà effettuata, ma serve solamente per orientare l'oggetto nel modo voluto prima di scalarlo. La sintassi è identica al metodo `rotation`. Ad esempio

  ```
  xform->scaleOrientation.setValue( 1.0, 0.0, 0.0, 1.5 );
  ```

  significa effettuare una rotazione, prima dello scalamento, intorno all'asse x di 1.5 radianti. Il valore di default alla creazione del nodo è $[0.0, 0.0, 1.0, 0.0]$.

- **center** definisce l'origine per scalamenti e rotazioni. Ad esempio

  ```
  xform->center.setValue( 1.0, 0.0, 0.0 );
  ```

  definisce l'origine delle trasformazioni nel punto $[1.0, 0.0, 0.0]$. Il valore di default alla creazione del nodo è $[0.0, 0.0, 0.0]$, l'origine del sistema di riferimento cartesiano.

Per usare questo nodo, includere il file header `Inventor/nodes/SoTransform.h`. L'ordine in cui i campi agiscono sull'oggetto è:

1. **center**,
2. **scaleOrientation**,

3. **scaleFactor**,
4. **rotation**,
5. **translation**.

Quindi per prima cosa viene settato l'origine delle trasformazioni, poi le direzioni dello scalamento, ed infine vengono effettuate nell'ordine le trasformazioni di scalamento, rotazione e traslazione.

Al posto del nodo **SoTransform**, si possono usare i seguenti nodi (classi derivate da **SoTransform**), ognuno rappresentante una trasformazione:

- **SoTranslation** rappresenta la traslazione lungo gli assi x, y e z. L'equivalente del codice del nodo **SoTransform** diventa:

  ```
  SoTranslation*  translation = new SoTranslation;

  translation->translation.setValue( 1.0, 2.0, -1.5 );
  ```

  Il valore di default alla creazione del nodo è $[0.0, 0.0, 0.0]$. Per usare questo nodo, includere il file header `Inventor/nodes/SoTranslation.h`.

- **SoRotation** specifica la rotazione intorno ad un asse qualunque, passante per l'origine, di un certo numero di gradi in radianti. Ad esempio

  ```
  SoRotation*  rotation = new SoRotation;

  rotation->rotation.setValue( 1.0, 0.0, 0.0, 1.5 );
  ```

  Se l'asse non passa per l'origine, va portato nell'origine con traslazioni della scena. Ovviamente, eseguita la rotazione, deve essere eseguita una traslazione opposta per rimettere l'asse nella posizione primitiva. Il valore di default alla creazione del nodo è $[0.0, 0.0, 1.0, 0.0]$. Per usare questo nodo, includere il file header `Inventor/nodes/SoRotation.h`.

- **SoRotationXYZ** specifica la rotazione intorno ad uno degli assi coordinati di un certo numero di gradi in radianti. Ad esempio

  ```
  SoRotationXYZ*  rotation = new SoRotationXYZ;

  rotation->axis = SoRotationXYZ::X;
  rotation->angle = M_PI/2;
  ```

  significa effettuare una rotazione intorno all'asse x di $\pi/2$. Il valore di default alla creazione del nodo è *axis* = `X` e *angle* $= 0$. Per usare questo nodo, includere il file header `Inventor/nodes/SoRotationXYZ.h`.

- **SoScale** effettua uno scaling lungo gli assi x, y e z. Ad esempio

  ```
  SoScale*  scale = new SoScale;

  scale->scaleFactor.setValue( 1.0, 2.0, 0.75 );
  ```

effettua uno scalamento lungo gli assi x, y e z rispettivamente di 1.0, 2.0 e 0.75. Il valore di default alla creazione del nodo è $[1.0, 1.0, 1.0]$. Per usare questo nodo, includere il file header `Inventor/nodes/SoScale.h`.

Nell'esempio seguente si vuole mettere in evidenza come l'ordine con cui compaiono i nodi di trasformazine nella scena sia fondamentale. All'oggetto di sinistra (`leftSep`) vengono applicate nell'ordine uno scalamento, una rotazione ed una traslazione. All'oggetto di destra (`rightSep`) vengono applicate le stesse trasformazioni, ma viene effettuata prima la rotazione, poi lo scalamento ed infine la traslazione.

```
/*--------------------------------------------------------------
 *  This is an example from the Inventor Mentor,
 *  chapter 5, example 6.
 *
 *  This example shows the effect of different order of
 *  operation of transforms.  The left object is first
 *  scaled, then rotated, and finally translated to the left.
 *  The right object is first rotated, then scaled, and finally
 *  translated to the right.
 *------------------------------------------------------------*/

#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/SoDB.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoRotationXYZ.h>
#include <Inventor/nodes/SoScale.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoTranslation.h>

int main( int, char** argv )
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);
   if(myWindow == 0) return 1;

   SoSeparator* root = new SoSeparator;
   root->ref();

   // Create two separators, for left and right objects.
   SoSeparator* leftSep = new SoSeparator;
   SoSeparator* rightSep = new SoSeparator;
   root->addChild(leftSep);
   root->addChild(rightSep);

   // Create the transformation nodes
   SoTranslation* leftTranslation  = new SoTranslation;
   SoTranslation* rightTranslation = new SoTranslation;
   SoRotationXYZ* myRotation = new SoRotationXYZ;
   SoScale* myScale = new SoScale;

   // Fill in the values
   leftTranslation->translation.setValue(-1.0, 0.0, 0.0);
   rightTranslation->translation.setValue(1.0, 0.0, 0.0);
   myRotation->angle = M_PI/2;   // 90 degrees
   myRotation->axis = SoRotationXYZ::X;
   myScale->scaleFactor.setValue(2., 1., 3.);

   // Add transforms to the scene.
   leftSep->addChild(leftTranslation);   // left graph
```

```
    leftSep->addChild(myRotation);        // then rotated
    leftSep->addChild(myScale);           // first scaled

    rightSep->addChild(rightTranslation); // right graph
    rightSep->addChild(myScale);          // then scaled
    rightSep->addChild(myRotation);       // first rotated

    // Read an object from file
    SoInput myInput;
    if(!myInput.openFile("temple.iv")) return 1;
    SoSeparator* fileContents = SoDB::readAll(&myInput);
    if(fileContents == 0) return 1;

    // Add an instance of the object under each separator.
    leftSep->addChild(fileContents);
    rightSep->addChild(fileContents);

    // Construct a renderArea and display the scene.
    SoXtExaminerViewer* myViewer = new SoXtExaminerViewer(myWindow);
    myViewer->setSceneGraph(root);
    myViewer->setTitle("Transform Ordering");
    myViewer->viewAll();
    myViewer->show();

    SoXt::show(myWindow);
    SoXt::mainLoop();

    return 0;
}
```