Open Inventor
2 - Struttura Gerarchica dei Nodi e Tools

Claudio Baldazzi *Maurizio Patrignani*

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

18 marzo 1998

1 Introduzione

Questi appunti hanno lo scopo di illustrare come si costruisce una scena 3D partendo da oggetti elementari, proprietà di visualizzazione e gruppi di oggetti. La scena all'interno di Open Inventor è rappresentata attraverso una struttura gerarchica di grafo (*scene graph*) i cui nodi rappresentano oggetti elementari o composti oppure proprietà di visualizzazione e di materiali degli oggetti. Si cercherà di approfondire ed evidenziare le regole che determinano la visita di uno scene graph, in special modo durante l'azione di rendering, che porta effettivamente a visualizzare una scena nel programma applicativo.

Infine si descriverà molto sommariamente la *Inventor Component Library*, in special modo la render area.

Le informazioni contenute in questo documento e i listati discussi, si trovano sul libro "The Inventor Mentor" di Josie Wernecke edito dalla Addison Wesley.

2 Scene Database (Database della Scena)

Lo *scene database* di Inventor (**SoDB**) è l'insieme delle informazioni rappresentanti una o più scene tridimensionali. Questo database può contenere più *scene graph*, come mostrato in figura 1. Il database in figura contiene 5 grafi, quindi 5 scene. Le scene nell'esempio rappresentano un'automobile, una piccola casa, un'altra automobile, una casa grande e una persona.

Dal punto di vista programmativo, il database non è altro che una classe con dei metodi, il più importante dei quali, **SoDB::init()**, serve ad inizializzare il database. Il database va sempre inizializzato per poter usare Inventor. Se però si usa la Inventor Component Library, questo non è sempre necessario, poiché certi comandi, come **SoXt::init()** inizializzano automaticamente il database. Il database può essere direttamente popolato da file con i metodi:

```
SoSeparator readAll( SoInput* in )
SbBool read( SoInput* in, SoNode* &rootNode )
SbBool read( SoInput* in, SoPath* &path )
```

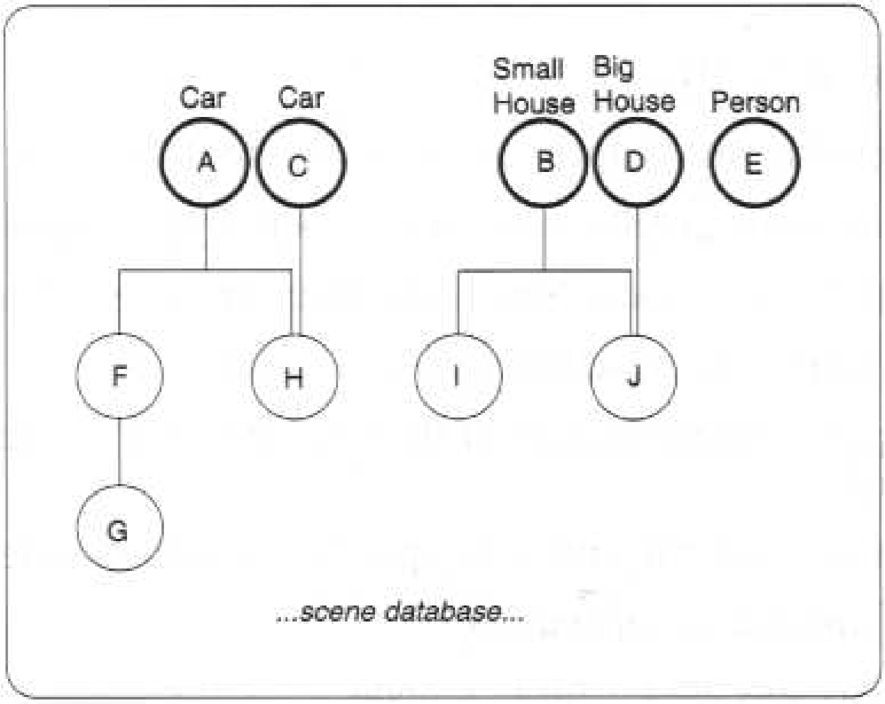

Figura 1: Un esempio di Database di Open Inventor

Il primo metodo corrisponde ad una funzione che legge i dati del file specificato in `in`, li inserisce nel database e ritorna un puntatore ad un nodo separatore che ha come figli tutte le root degli scene graph. Il secondo e terzo prototipo leggono il file da `in`, inseriscono i dati nel database e restituiscono il nodo di root o il path risultante di una scena.

3 Scene Graph (Grafo della Scena) e Tipi di Nodo

Uno *scene graph* è costituito da uno o più nodi, ognuno dei quali rappresenta un oggetto primitivo, una proprietà o un gruppo di nodi. La gerarchia della scena è creata addizionando nodi come figli di altri nodi. Il risultato è un *grafo diretto aciclico*. Il primo nodo di uno scene graph è chiamato *root node* (nodo di root), o semplicemente root. Nell'esempio di figura 1 i nodi A, C, B, D ed E sono nodi root. In generale un nodo può avere anche più padri, cioè può essere instanziato da più nodi, ad esempio i nodi H e J dell'esempio in figura 1. Un nodo di questo tipo si chiama *shared instancing node.*

Un nodo quindi è l'elemento fondamentale del grafo della scena. Esso contiene dati e metodi che definiscono oggetti 3D, proprietà o gruppi di nodi. Ogni nodo in effetti è un oggetto istanziato di una classe C++, quindi con i suoi dati e metodi. Quando si crea un nodo, esso è automaticamente inserito nel database come un nodo root. In realtà poi si connette il nodo ad altri nodi creando una struttura gerarchica.

Per creare un nodo bisogna usare sempre l'operatore **new** del C++, ad esempio

```
SoSphere*  headSphere = new SoSphere;
```

Un nodo non può essere cancellato con l'operatore **delete**, poiché, come si vedrà meglio più avanti, è Inventor stesso che cancella gli oggetti dal proprio database.

I nodi di Open Inventor sono divisibili in tre fondamentali categorie:

- *Shape nodes*, che rappresentano oggetti primitivi 3D;
- *Property nodes*, che rappresentano caratteristiche qualitative della scena;
- *Group nodes*, che rappresentano collezioni di nodi nel grafo.

3.1 Shape Nodes

Gli *shape nodes* rappresentano oggetti primitivi 3D, quali cubi, coni, cilindri, testi, curve e superfici. Essi descrivono l'oggetto grafico dal punto di vista fisico, punti e linee, e sono influenzati dai *property nodes* e *group nodes*, e, durante l'azione di rendering, essi disegnano loro stessi sullo schermo. In figura 2 è

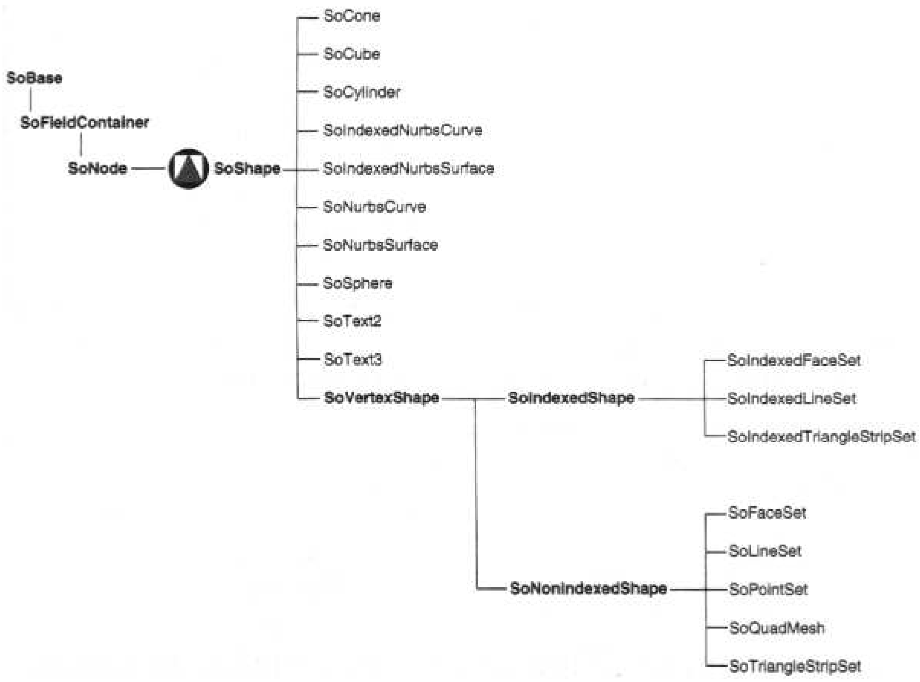

Figura 2: Gerarchia di classi degli *shape nodes*

visibile la gerarchia di classi C++ relativa a questi nodi.

3.2 Property Nodes

I *property nodes* rappresentano l'apparenza e le caratteristiche qualitative della scena, come il materiale degli oggetti, lo stile di disegno o le trasformazioni geometriche. Questo tipo di nodi possono essere divisi in tre sottocategorie:

- Nodi *transform*, che realizzano trasformazioni geometriche, quali **SoTransform**, **SoRotation**, **SoScale**, **SoTranslation**, **SoRotationXYZ**. Tutti questi nodi sono derivati da **SoTransformation**. Ad esempio:

  ```
  SoTransform*  myXform = new SoTransform;

  // set field value
  myXform->translation.setValue( 0.0, -1.0, 0.0 );
  ```

- Nodi *appearance*, che modificano l'apparenza degli oggetti, quali **SoMaterial**, **SoMaterialBinding**, **SoBaseColor**, **SoDrawStyle**, **SoLightModel**, **SoFont**. Ad esempio:

  ```
  SoMaterial*  bronze = new SoMaterial;

  // set field value
  bronze->ambientColor.setValue( .33, .22, .27 );
  bronze->diffuseColor.setValue( .78, .57, .11 );
  bronze->specularColor.setValue( .99, .94, .81 );
  bronze->shininess = .28;
  ```

- Nodi *metrics*, che contengono coordinate, normali e altre informazioni geometriche per modificare certe caratteristiche degli oggetti. Esempi di tali nodi sono **SoCoordinate3**, **SoCoordinate4**, **SoNormal**.

In generale, un *property node* **rimpiazza** il corrispondente elemento nel *traversal state* (vedi dopo) con il proprio nuovo valore. Unica eccezione a questa regola sono le trasformazioni geometriche. Esse si **concatenano** con la trasformazione corrente.

3.3 Group Nodes

I *group nodes* sono dei contenitori di oggetti figli. Esistono diversi classi di *group nodes*, ognuna con caratteristiche di gruppo particolari.

Quando un nodo di questo tipo viene creato, esso non ha figli. La classe base per tutti i nodi Group è **SoGroup**, e tutti i nodi derivati da essa hanno il metodo `addChild()` per inserire dei nodi figli.

Supponendo di voler comporre il nodo `myXform` e il nodo `bronze`, definiti prima, e uno shape node sfera in un singolo gruppo, il gruppo "head", dobbiamo per prima cosa creare un **SoGroup** e poi usare il metodo `addChild()` per ogni nodo da associare al gruppo:

```
SoGroup*  head = new SoGroup;

head->addChild(myXform);
head->addChild(bronze);
head->addChild(new SoSphere);
```

Il metodo `addChild()` inserisce un nodo alla fine della lista di figli nel gruppo. Ogni figlio inserito nel gruppo ha associato un indice. Il primo figlio del gruppo ha indice 0, il secondo ha indice 1 e così via. L'ordine con cui i nodi sono inseriti in un gruppo è fondamentale quando si applica al database una azione come la *rendering action*.

Nell'esempio seguente, di cui possiamo esaminare il grafo della scena in figura 3, si può osservare la costruzione di un oggetto complesso, raggruppando gli elementi di base. L'oggetto in questione rappresenta la molecola dell'acqua. Si

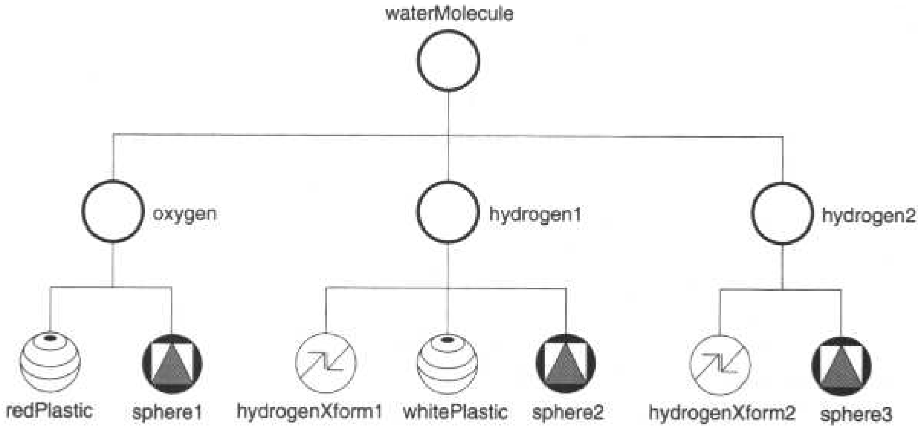

Figura 3: Scene Graph relativo alla molecola dell'acqua

costruisce intanto l'atomo di ossigeno, che sarà un gruppo **SoGroup** formato da un nodo che darà l'aspetto rosso ad una sfera. Si costruisce poi il primo atomo di idrogeno, composto da un altro gruppo con tre nodi: un nodo che setta l'apparenza degli oggetti al bianco, una trasformazione che porta il primo atomo di idrogeno alla posizione corretta, infine un nodo rappresentante la sfera. Come ultimo passo si costruisce il secondo atomo di idrogeno, ereditando il materiale definito per il primo atomo di idrogeno. Questo gruppo sarà quindi formato da due soli nodi, uno di trasformazione che si concatena con i precedenti, e l'altro di tipo *shape* rappresentante la sfera.

```
/*--------------------------------------------------------------
 *  This is an example from the Inventor Mentor,
 *  chapter 3, example 1.
 *
 *  This code shows how to create a molecule out of 3 spherical
 *  atoms.  The molecule illustrates how the ordering of nodes
 *  within a scene graph affects the rendered image.
 *
 *------------------------------------------------------------*/

#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/nodes/SoGroup.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/nodes/SoTransform.h>

SoGroup* makeWaterMolecule()
```

```
{
   // Construct all parts
   SoGroup* waterMolecule = new SoGroup;  // water molecule

   SoGroup* oxygen = new SoGroup;         // oxygen atom
   SoMaterial* redPlastic = new SoMaterial;
   SoSphere* sphere1 = new SoSphere;

   SoGroup* hydrogen1 = new SoGroup;      // hydrogen atoms
   SoGroup* hydrogen2 = new SoGroup;
   SoTransform* hydrogenXform1 = new SoTransform;
   SoTransform* hydrogenXform2 = new SoTransform;
   SoMaterial* whitePlastic = new SoMaterial;
   SoSphere* sphere2 = new SoSphere;
   SoSphere* sphere3 = new SoSphere;

   // Set all field values for the oxygen atom
   redPlastic->ambientColor.setValue( 1.0, 0.0, 0.0 );
   redPlastic->diffuseColor.setValue( 1.0, 0.0, 0.0 );
   redPlastic->specularColor.setValue( 0.5, 0.5, 0.5 );
   redPlastic->shininess = 0.5;

   // Set all field values for the hydrogen atoms
   hydrogenXform1->scaleFactor.setValue( 0.75, 0.75, 0.75 );
   hydrogenXform1->translation.setValue( 0.0, -1.2, 0.0 );
   hydrogenXform2->translation.setValue( 1.1852, 1.3877, 0.0 );
   whitePlastic->ambientColor.setValue( 1.0, 1.0, 1.0 );
   whitePlastic->diffuseColor.setValue( 1.0, 1.0, 1.0 );
   whitePlastic->specularColor.setValue( 0.5, 0.5, 0.5 );
   whitePlastic->shininess = 0.5;

   // Create a hierarchy
   waterMolecule->addChild(oxygen);
   waterMolecule->addChild(hydrogen1);
   waterMolecule->addChild(hydrogen2);

   oxygen->addChild(redPlastic);
   oxygen->addChild(sphere1);
   hydrogen1->addChild(hydrogenXform1);
   hydrogen1->addChild(whitePlastic);
   hydrogen1->addChild(sphere2);
   hydrogen2->addChild(hydrogenXform2);
   hydrogen2->addChild(sphere3);

   return waterMolecule;
}

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();

   // This function contains our code fragment.
   root->addChild(makeWaterMolecule());

   SoXtExaminerViewer* myViewer = new SoXtExaminerViewer(myWindow);
   myViewer->setSceneGraph(root);
   myViewer->setTitle("H two O");
```

```
   myViewer->show();
   myViewer->viewAll();

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

Si noti come l'ordine degli oggetti sia fondamentale, infatti il secondo atomo di idrogeno eredita tutto dal primo, anche la trasformazione. La trasformazione da applicare al secondo atomo, va quindi calcolata in base alla precedente, poiché esse si concatenano.

Di seguito si descriverà in dettaglio come avviene la visita del grafo di questa scena.

3.4 Nodo Separator e di root

Per isolare l'effetto dei nodi in un gruppo si usa il nodo **SoSeparator**. Prima di visitare i suoi figli, un nodo **SoSeparator** salva lo stato corrente della visita. Quando ha finito di visitare i propri figli, il nodo **SoSeparator** ripristina lo stato che aveva salvato. Quindi i nodi all'interno di un **SoSeparator** non hanno influenza sui nodi a destra del grafo.

Il nodo root di un grafo, dovrebbe sempre essere un **SoSeparator** per resettare lo stato tra successive rendering action.

Il prossimo esempio evidenzia l'uso dei nodi **SoSeparator** nella costruzione di un piccolo semplice robot. La struttura del grafo della scena è visibile in figura 4.

```
/*--------------------------------------------------------------
 *  This is an example from the Inventor Mentor,
 *  chapter 3, example 2.
 *
 *  This code shows how to create a robot out of various nodes.
 *  It introduces shared instancing of nodes to create two legs
 *  using two instances of the same subgraph.
 *
 *------------------------------------------------------------*/
//--------------------------------------------------------------
//  Modificato per strutturare le parti che compongono il robot.
//--------------------------------------------------------------

#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoCylinder.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/nodes/SoTransform.h>

// Local Function Prototyping
SoSeparator* makeLeg();
SoSeparator* makeBody();
SoSeparator* makeHead();
SoSeparator* makeRobot();

int main( int, char** argv )
{
    // Initialize Inventor and Xt
```

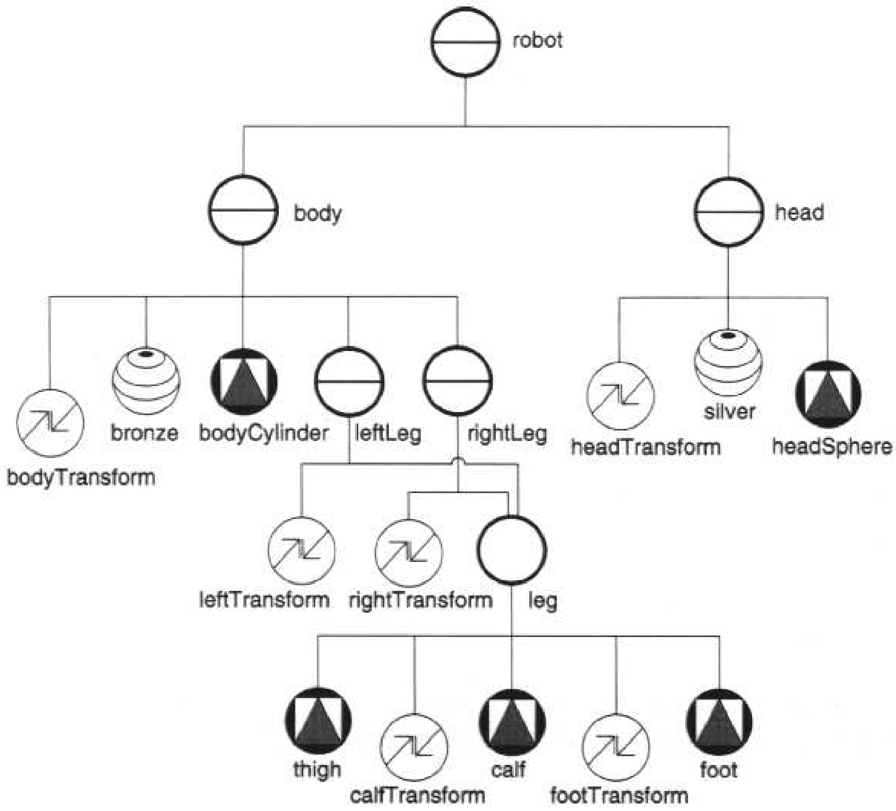

Figura 4: Scene Graph relativo al robot

```
    Widget myWindow = SoXt::init(argv[0]);
    if( myWindow == 0 ) return 1;

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

    // This function contains our code fragment.
    root->addChild(makeRobot());

    SoXtExaminerViewer* myViewer = new SoXtExaminerViewer(myWindow);
    myViewer->setSceneGraph(root);
    myViewer->setTitle("Robot");
    myViewer->show();
    myViewer->viewAll();

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

    return 0;
}

SoSeparator* makeRobot()
{
    // Robot is just head and body
    SoSeparator* robot = new SoSeparator;
```

```
    robot->addChild(makeBody());
    robot->addChild(makeHead());

    return robot;
}

SoSeparator* makeBody()
{
    // Parts for body
    SoTransform* bodyTransform = new SoTransform;
    bodyTransform->translation.setValue( 0.0, 3.0, 0.0 );

    SoMaterial* bronze = new SoMaterial;
    bronze->ambientColor.setValue( .33, .22, .27 );
    bronze->diffuseColor.setValue( .78, .57, .11 );
    bronze->specularColor.setValue( .99, .94, .81 );
    bronze->shininess = .28;

    SoCylinder* bodyCylinder = new SoCylinder;
    bodyCylinder->radius = 2.5;
    bodyCylinder->height = 6;

    // Add legs
    SoSeparator* leg = makeLeg();

    // Left leg
    SoTransform* leftTransform = new SoTransform;
    leftTransform->translation = SbVec3f( 1, -4.25, 0 );
    SoSeparator* leftLeg = new SoSeparator;
    leftLeg->addChild(leftTransform);
    leftLeg->addChild(leg);

    // Right leg
    SoTransform* rightTransform = new SoTransform;
    rightTransform->translation.setValue( -1, -4.25, 0 );
    SoSeparator* rightLeg = new SoSeparator;
    rightLeg->addChild(rightTransform);
    rightLeg->addChild(leg);

    // Construct body out of parts
    SoSeparator* body = new SoSeparator;
    body->addChild(bodyTransform);
    body->addChild(bronze);
    body->addChild(bodyCylinder);
    body->addChild(leftLeg);
    body->addChild(rightLeg);

    return body;
}

SoSeparator* makeHead()
{
   // Head parts
   SoTransform* headTransform = new SoTransform;
   headTransform->translation.setValue( 0, 7.5, 0 );
   headTransform->scaleFactor.setValue( 1.5, 1.5, 1.5 );

   SoMaterial* silver = new SoMaterial;
   silver->ambientColor.setValue( .2, .2, .2 );
   silver->diffuseColor.setValue( .6, .6, .6 );
   silver->specularColor.setValue( .5, .5, .5 );
```

```
   silver->shininess = .5;

   SoSphere* headSphere = new SoSphere;

   // Construct head
   SoSeparator* head = new SoSeparator;
   head->addChild(headTransform);
   head->addChild(silver);
   head->addChild(headSphere);

   return head;
}

SoSeparator* makeLeg()
{
    // Construct parts for legs (thigh, calf and foot)
    SoCube* thigh = new SoCube;
    thigh->width = 1.2; thigh->height = 2.2; thigh->depth = 1.1;

    SoTransform* calfTransform = new SoTransform;
    calfTransform->translation.setValue( 0, -2.25, 0.0 );

    SoCube* calf = new SoCube;
    calf->width = 1; calf->height = 2.2; calf->depth = 1;

    SoTransform* footTransform = new SoTransform;
    footTransform->translation.setValue( 0, -1.5, .5 );

    SoCube* foot = new SoCube;
    foot->width = 0.8; foot->height = 0.8; foot->depth = 2;

    // Put leg parts together
    SoSeparator* leg = new SoSeparator;
    leg->addChild(thigh);
    leg->addChild(calfTransform);
    leg->addChild(calf);
    leg->addChild(footTransform);
    leg->addChild(foot);

    return leg;
}
```

4 Applicazione di una *action* ad un nodo

Ogni tipo di nodo implementa la sua specifica risposta ad ogni tipo di azione intrapresa sul grafo della scena. Quando si applica una azione sulla scena, essa si applica alla root del grafo della scena. Per ogni azione, il database di Inventor mantiene un *traversal state* (stato della visita), che altro non è che un insieme di elementi e parametri attuali della visita del grafo. La visita del grafo, a seguito di un'azione, avviene dall'alto al basso e da sinistra a destra. Durante la visita, i nodi possono modificare il *traversal state*, dipendentemente dal loro particolare tipo.

In questa sezione si parlerà della *OpenGL rendering action*.

4.1 OpenGL Rendering Action

Questa azione particolare ha lo scopo di disegnare sullo schermo la scena. I principali elementi del *traversal state* sono:

- Current geometric transformation
- Current material components
- Current lighting model
- Current drawing style
- Current text font
- Current coordinates
- Current normals
- Current lights
- Current viewing specification

Tutti i nodi che modificano il *traversal state*, sostituiscono il vecchio valore con il nuovo, eccetto le trasformazioni, che si concatenano con quelle del *traversal state*.

In particolare, la rendering action, sui nodi si comporta nel seguente modo:

- Se il nodo incontrato nella visita è un *group node*, esso invocherà la rendering action su ognuno dei propri figli, in ordine.
- Ogni nodo figlio, a turno eseguirà il proprio metodo di rendering, che può modificare il *traversal state*. Se il nodo è di tipo property node allora esso modificherà il *traversal state*, rimpiazzando il valore precedente oppure concatenandosi con il valore precedente nel caso di trasformazioni geometriche.
- Se il nodo è uno *shape node*, allora esso disegna se stesso usando l'attuale *traversal state*.

Nell'esempio di figura 3, durante la rendering action si visita il primo nodo, quello di root chiamato nel programma `waterMolecule`. Quando a questo nodo è applicata l'azione, esso visita il proprio primo figlio, oxygen. Questo gruppo a sua volta applica l'azione sui suoi figli nel seguente modo:

1 Il nodo `redPlastic` cambia il tipo di materiale a plastica rossa.

2 Il nodo `sphere1` causa il disegno della sfera secondo lo stato corrente e una sfera rossa viene disegnata nell'origine.

La visita del grafo continua ora sul gruppo `hydrogen1`.

3 Il nodo di trasformazione `hydrogenXform1` modifica la matrice di trasformazione corrente inserendo uno scalamento ed una traslazione.

4 Il nodo `whitePlastic` cambia il tipo di materiale a plastica bianca.

5 Il nodo `sphere2` causa il disegno di un'altra sfera secondo lo stato corrente e una sfera bianca, piú piccola a causa dello scalamento, viene disegnata vicino la sfera rossa.

Di seguito viene visitato il gruppo `hydrogen2`.

6 Il nodo di trasformazione `hydrogenXform2` modifica la matrice di trasformazione corrente inserendo una traslazione.

7 Il nodo `sphere3` causa il disegno di un'altra sfera secondo lo stato corrente e una sfera bianca, piú piccola a causa dello scalamento precedente, viene disegnata vicino la sfera rossa, dalla parte opposta dell'altra sfera bianca.

5 Reference e Deletion di nodi

Come abbiamo già constatato, i nodi vengono creati attraverso i costrutti standard del C++ (**new**), mentre non possono essere cancellari attraverso la **delete** del C++. Il perché di questo sarà chiarito in questa sezione.

Ogni nodo memorizza, all'interno del database, il numero di riferimenti fatti a se stesso. I riferimenti al nodo, quindi l'incremento del contatore, avvengono quando si crea un link tra un nodo padre e un nodo figlio (ad esempio l'istruzione `node->addChild(child)` incrementa di uno il contatore del nodo `child`), quando si crea un cammino (*path*), quando avviene un'azione, etc., la tabella 1 riassume le azioni che fanno aumentare o diminuire il reference count. Quando il reference count di un nodo, a seguito di un'azione sul database, va a 0, il nodo viene cancellato. Per questo è importante aumentare il reference count del nodo di root con l'istruzione `root->ref()`. Quando il nodo di root viene creato ha, come tutti i nodi, il reference count pari a 0. Siccome, in genere, al nodo di root non viene applicata un'azione che fa incrementare in maniera permanente il reference count, alla prima azione applicata al database esso verrebbe cancellato. Per tutti questi motivi, è anche necessario creare i nodi sempre in memoria dinamica (con **new**) e non sullo stack come variabili locali di una funzione.

Tabella 1: References and Deletion.

Increments Reference Count by 1	Decrements Reference Count by 1
Adding a node as a child of another node increments child's reference count	Removing a node as a child of another node
Adding a node to a path	Removing a node from a path
Applyng an action to a node or path increments reference count of all nodes that are traversed	When traversal for the action finishes, all nodes that were traversed are unreferenced
Adding a node to an SoNodeList node	Removing a node from an SoNodeList
Settng an SoSFNode or SoMFNode value to point to a node	Changing an SoSFNode or SoMFNode value to point to a different node or to NULL, or deleting the value
Connecting an output of an engine to a field in a node or engine increments the engine's reference count	Disconnecting an engine's output from the field decrements the engine's reference count

6 L'*Inventor Component Library*

Per la gestione delle finestre e dell'interfaccia utente Open Inventor offre al programmatore l'*Inventor Component Library*. Quest'ultima è composta di tre parti:

- Le funzioni della `Xt utility` per l'inizializzazione e la gestione delle finestre e dell'ambiente (per esempio le funzioni `SoXt::init()`, `SoXt::show()`, ecc).
- La speciale classe `SoXtRenderArea` per la visualizzazione di uno scene graph.
- Un insieme di componenti Xt (`Xt components`), ognuno dei quali contiene la propria render area ed una interfaccia utente per modificare e navigare nella scena rappresentata.

6.1 Le funzioni dell'*Xt Utility* e la `SoXtRenderArea`

In generale non è detto che un'applicazione che faccia uso di Open Inventor debba necessariamente visulizzare qualcosa sullo schermo, anche se ciò è molto probabile. Una applicazione che visualizza oggetti tridimensionali sullo schermo ricorre sempre, direttamente o indirettamente, ad una `SoXtRenderArea`, ed utilizza sempre le funzioni dell'*Xt Utility* per l'inizializzazione del sistema, la gestione degli eventi ecc. Se l'applicazione gestisce direttamente una `SoXtRenderArea`, essa dovrà eseguire necessariamente le operazioni seguenti:

1. Inizializzare Inventor per essere usato con le Xt Intrisics (`SoXt::init()`).
2. Creare la `SoXtRenderArea`.
3. Costruire una scena (ed eventualmente altri widget se il programma lo richiede).
4. Mostrare la render area e gli altri widget Xt (`myRenderArea->show(); SoXt::show()`).

Il programma seguente esegue tutte queste operazioni in successione e si può considerare uno scheletro, modificabile dal programmatore a seconda delle esigenze.

```
#include <X11/Intrinsic.h>
#include <Inventor/So.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>

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

  SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);

  SoSeparator *root = new SoSeparator;
  // Build other Inventor objects and Xt widgets
  // and set up the root
```

```
  // ...

  myRenderArea->setSceneGraph(root);
  myRenderArea->setTitle("Simple Xt");
  myRenderArea->show(); // this calls XtManageChild
  Soxt::show(myWindow); // this calls XtRealizeWidget

  // Realize other Xt widgets
  // ...

  // Go into main event loop
  SoXt::mainLoop();
}
```

Il costruttore della `SoXtRenderArea` è il seguente:

```
SoXtRenderArea(Widget parent = NULL,
               const char * name = NULL,
               SbBool buildInsideParent = TRUE,
               SbBool getMouseInput = TRUE,
               SbBool getKeyboardInput = TRUE);
```

Alcuni metodi della `SoXtRenderArea` di uso più frequente sono i seguenti:

- `setBorder()` mostra o nasconde i bordi della finestra.
- `setBackgroundColor()` specifica il colore dello sfondo della finestra.
- `setAutoRedraw()` abilita o disabilita il sensore che ridisegna automaticamente lo schermo quando necessario.
- `render()` ridisegna la scena immediamtamente. Se `AutoRedraw` è posto a `TRUE` non è necessario invocare questa funzione.

7 I *Components*

La maggior parte delle applicazioni necessitano di interfacce grafiche sofisticate, basate sulla `SoXtRenderArea`, ma che consentano all'utente di interagire con la scena rappresentata (per esempio: cambiare il punto di vista, zoomare, modificare i parametri dei nodi, ...). Per non costringere il programmatore a costruirsi una intera interfaccia a partire dalla `SoXtRenderArea`, quando non abbia esigenze troppo particolari, Open Inventor offre gli *Xt Components*. Essi cadono in due grandi classi: i *viewers*, per la visualizzazione, e gli *editors*, per la modifica della scena rappresentata.

Esempi di *viewers* sono il `SoXtExaminerViewer` (già visto nella prima esercitazione), il `SoXtFlyViewer` (permette di "volare" intorno all'oggetto), il `SoXtWalkViewer` (permette di "camminare" intorno all'oggetto), ecc. Esempi di *editors* sono il `SoXtMaterialEditor` (un interfaccia grafica per modificare i parametri di un nodo `SoMaterial`), il `SoXtDirectionalLightEditor` (per modificare i parametri di un nodo `SoDirectionalLight`), ecc.

Per utilizzare un *component* occorre:

- crearlo con l'opportuno costruttore, specificando il nome, il widget genitore, e se il *component* debba albergare all'interno del widget genitore o meno.

- farlo apparire o scomparire dal video.
- passare i dati dal *component* all'applicazione.

Tutti i *component* hanno un costruttore analogo a quello della `SoXtRenderArea`:

```
SoXtComponent(Widget parent = NULL,
              const char * name = NULL,
              SbBool buildInsideParent = TRUE,
              SbBool getMouseInput = TRUE,
              SbBool getKeyboardInput = TRUE);
```

Alcuni metodi della classe `SoXtComponent` di uso più frequente sono i seguenti:

- `show()` mostra il *component*.
- `hide()` nasconde il *component*.
- `setTitle()` pone una stringa sulla title bar del *component*.
- `setSize()` dimensiona il *component*.
- `getSize()` restituisce la dimensione del *component*.
- `isVisible()` restituisce `TRUE` se il *component* è attualmente mostrato sullo schermo.

Il passaggio dei dati tra un *component* e l'applicazione può avvenire in due modi:

1. tramite la definizione di *callback*, cioè funzioni che vengono attivate quando si verifica un evento nella finestra del *component* (per esempio: pressione o rilascio di un pulsante del mouse) e che vanno a modificare opportunamente alcune variabili o parametri dell'applicazione (non approfondiremo qui questo argomento).
2. tramite l'associazione del *component* ad un nodo. Questo è l'unico metodo ammesso per i *viewers*, ai quali si associa la radice dell'intero *scene graph* con la funzione `setSceneGraph()`. Anche gli *editors* vengono quasi sempre associati ad un nodo: per esempio il `SoXtMaterialEditor` viene associato ad un nodo `SoMaterial`, ed il `SoXtDirectionalLightEditor` viene associato ad un nodo di tipo `SoDirectionalLight`. Quando si vuole modificare i parametri di più nodi contemporaneamente si è costretti a ricorrere alle *callback*.