# Open Inventor

8 - Engines

*Claudio Baldazzi* *Maurizio Patrignani*

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

20 maggio 1998

## 1 Introduzione

Gli *engines* sono oggetti di Inventor che si possono connettere ai campi di altri nodi per creare animazioni di parti di scena e vincoli su alcuni elementi della scena. Gli engines possono essere connessi tra loro così da reagire l'uno con l'altro ed eventualmente creare animazioni sincrone tra più parti della scena.

Un semplice esempio può essere quello del mulino a vento visto nella prima lezione. Nel grafo della scena viene inserito un engine che descrive la rotazione delle pale del mulino eseguendo una rotazione incrementale delle pale in relazione al tempo trascorso. Un esempio più complesso potrebbe prevedere due engine collegati insieme: un'automobile la cui animazione è controllata da un engine e un secondo engine potrebbe controllare il movimento della macchina attraverso un collegamento con l'altro engine e contemporaneamente controllare una camera che segue l'automobile in corsa.

Le informazioni, le figure e i listati contenuti in questo documento, sono tratti, salvo diversa indicazione, dal libro "The Inventor Mentor" di Josie Wernecke edito dalla Addison Wesley.

## 2 Overview sugli Engines

Gli engines vengono generalmente usati in due differenti modi:

- per animare parti della scena;
- per vincolare una parte della scena in relazione ad altri oggetti della scena.

Si potrebbe pensare ad un engine come ad una scatola nera che riceve in input dei valori, esegue alcune operazioni su di essi e produce una serie di valori di output. I valori di input e quelli di output sono direttamente connessi ad alcuni campi di altri nodi del grafo di scena, quindi l'output prodotto da un engine verrà copiato nei campi dei nodi relativi. Un engine, si veda figura 1, ha l'input derivato dal tipo **SoField** e l'output derivato dalla classe **SoEngineOutput**.

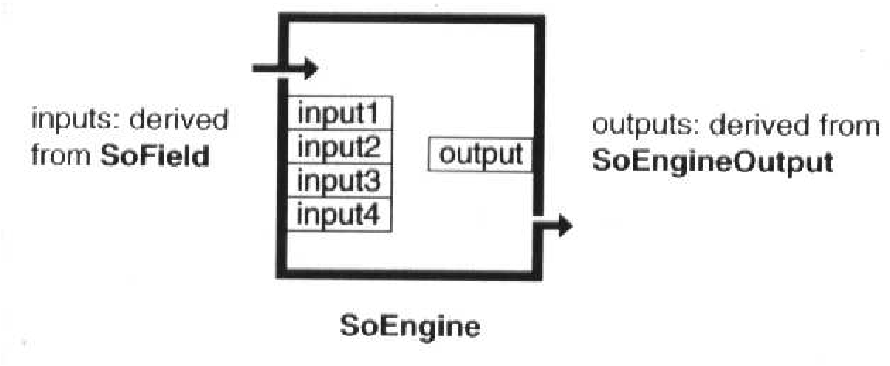


Figura 1: Anatomia di un engine

Ogni engine valuta una sua specifica funzione interna ogni qualvolta i valori di input cambiano e trasmette i valori di output a tutti i campi o altri engine connessi con esso. Siccome un nodo engine, **SoEngine**, deriva da un nodo **SoBase**, esso include i metodi per essere letto e scritto su file Inventor.

**Tipologie di engine** La figura 2 mostra l'albero delle classi per i nodi engine. Essi possono essere classificati secondo le seguenti categorie:

- Engine aritmetici:
  - **SoCalculator**
  - **SoBoolOperation**
  - **SoInterpolateFloat, SoInterpolateRotation, SoInterpolate-Vec2f, SoInterpolateVec3f, SoInterpolateVec4f**
  - **SoTransformVec3f**
  - **SoComposeVec2f, SoDecomposeVec2f, SoComposeVec3f, So-DecomposeVec3f, SoComposeVec4f, SoDecomposeVec4f, So-ComposeRotation, SoDecomposeRotation, SoComposeMatrix, SoDecomposeMatrix**
  - **SoComputeBoundingBox**
- Engine per animazioni:
  - **SoElapsedTime**
  - **SoOneShot**
  - **SoTimeCounter**
- Triggered Engine (engine "a scatto"):
  - **SoCounter**
  - **SoOnOff**
  - **SoTriggerAny**
  - **SoGate**

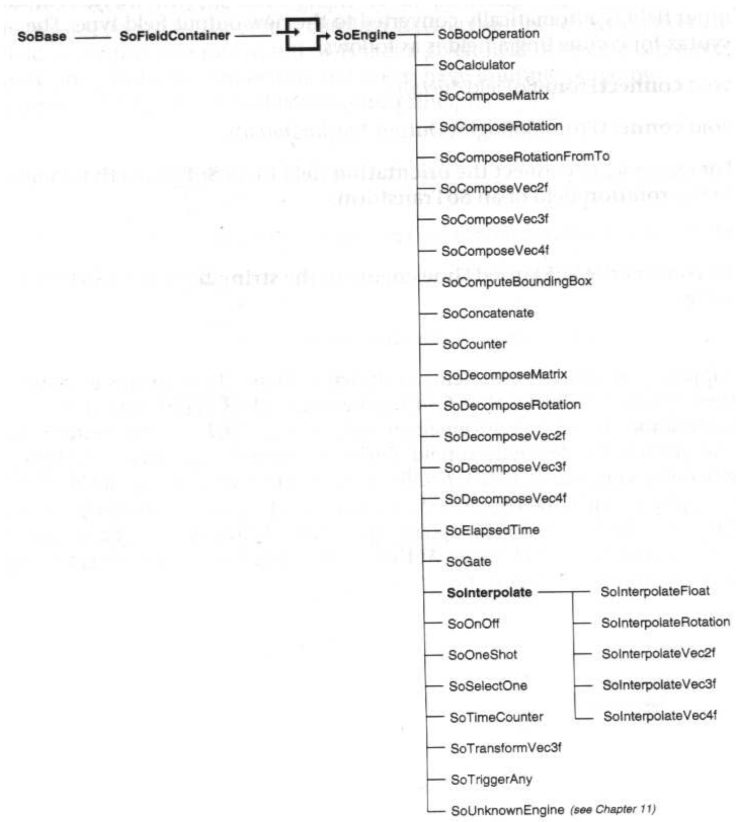


Figura 2: Albero degli engine

- Engine usati per la manipolazione di array:
  - **SoSelectOne**
  - **SoConcatenate**

# 3 Connessione ai campi

Per connettere un campo di un nodo ad un altro campo o ad un engine, si usa il metodo `connectFrom()` della classe **SoField**. Se vengono connessi campi di tipo differente, il valore del campo di input è automaticamente convertito nel tipo del campo di output. La sintassi del metodo è la seguente, a seconda del tipo di connessione da effettuare, se con un altro campo o con un engine:

```
void connectFrom( SoField* field );
void connectFrom( SoEngineOutput* engineOutput );
```

Per esempio, per connettere il campo *orientation* del nodo **SoPerspective** con il campo *rotation* del nodo **SoTransform**:

```
xform->rotation.connectFrom( &pCamera->orientation );
```

Per connettere l'engine **SoElapsedTime** con il campo *string* del nodo **SoText3**:

```
yourText->string.connectFrom( &elapsedTime->timeOut );
```

Si pensi ora di connettere reciprocamente i due campi di due oggetti. Ammettiamo che il *fieldA* del primo oggetto, sorgente, è connesso con il *fieldB* del secondo oggetto, destinazione, e viceversa, cioè il *fieldB*, sorgente, è connesso con il *fieldA*, destinazione. Una volta instaurata questa connessione, ogni volta che il *fieldA* viene cambiato, automaticamente anche il *fieldB* cambia e ogni volta che il *fieldB* viene cambiato, cambia il *fieldB*. Si può pensare che in questo modo si genera un loop infinito in cui i due campi si aggiornano l'uno con l'altro. Nell'implementazione attuale di Inventor questo non accade. Quando il *fieldA* cambia, cambia anche il *fieldB*, ma ora il *fieldA* sa che è già cambiato e non cambia di nuovo.

Si usa il metodo `disconnect()` per far terminare una connessione (sul nodo di destinazione) ed il metodo `isConnected()` per verificare se una connessione esiste oppure no.

Se si usa il metodo `connectFrom()` su un campo o un engine che ha già una connessione da una sorgente differente, la connessione originale viene eliminata e sostituita con la nuova.

Quando vengono connessi campi di diverso tipo, Inventor effettua automaticamente alcune conversioni:

- Qualsiasi tipo di campo a **String** e da **String** a qualsiasi tipo di campo.
- Tra qualsiasi dei seguenti due tipi: **Bool**, **Float**, **Long**, **Short**, **ULong**, **UShort**.
- Tra **Color** e **Vec3f**.
- Tra **Float** e **Time**.
- Tra **matrix** e **Rotation**.
- Tra **Name** e **Enum**.
- Tra **Rotation** e **Vec4f**.
- Tra un campo **MF** alla sua versione **SF** e viceversa.

Ogni volta che l'output di un engine è collegato ad un campo di un nodo, il *reference count* dell'engine viene incrementato di uno. Similmente rimuovendo una connassione tra un campo e l'output di un engine il suo *reference count* viene decrementato di 1. Quando anche l'ultima connessione dell'engine viene eliminata, il *reference count* dell'engine è 0 e l'engine viene cancellato dal database di Inventor. Le connessioni tra campi o le connessioni tra l'input di un engine ed un campo non modificano il *reference count* dell'engine. Per non modificare il *reference count* dell'engine, si può temporaneamente disabilitare la connessione richiamando il metodo `enableConnection(FALSE)` sul nodo destinazione oppure richiamando il metodo `enable(FALSE)` sull'output dell'engine.

# 4 Campi Globali

I campi globali (*global fields*) sono dei campi nel database di Inventor a cui si può accedere attraverso un nome e non sono contenuti in nessun particolare nodo o engine. Un esempio di *global field* built-in in Inventor è il **realTime**, di tipo SoSFTime. Questo campo contiene l'ora corrente e può essere connesso ad altri campi o engine per produrre animazioni temporizzate. Un campo globale può essere creato dal programmatore con il metodo `createGlobalField()` della classe **SoDB**:

```
static SoField* SoDB::createGlobalField( SbName& n, SoType t );
```

Ci può essere solo un campo globale con un dato nome. Se il nome già esiste ed il campo esistente ha lo stesso tipo di quello che si vuole creare, esso è ritornato dal metodo, altrimenti il metodo ritorna 0. Il metodo `getGlobalField()` ritorna il campo dato un certo nome:

```
static SoField* SoDB::getGlobalField( const SbName& name );
```

L'esempio seguente crea un orologio digitale che connette una stringa **SoText3** al global fild **realTime**. Il grafo della scena è visibile in figura 3.

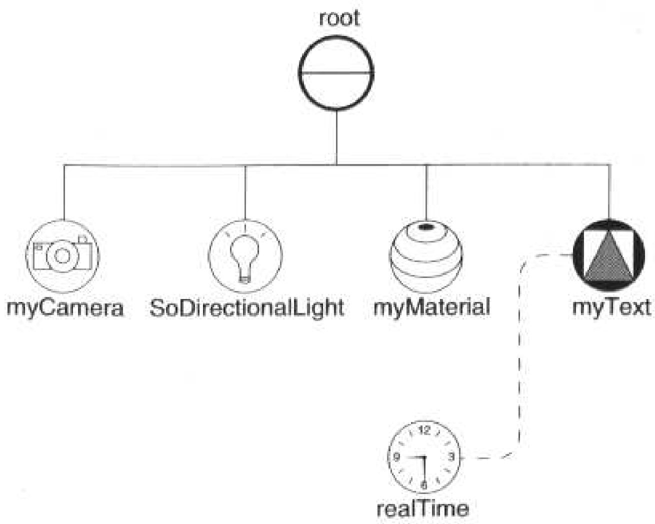


Figura 3: Grafo della scena dell'esempio con il campo globale realTime

```
/*--------------------------------------------------------------
 *  This is an example from the Inventor Mentor
 *  chapter 13, example 2.
 *
 *  Global fields.
 *  A digital clock is implemented by connecting the realTime
 *  global field to a Text3 string.
 *------------------------------------------------------------*/

#include <Inventor/SoDB.h>
```

```
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoText3.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();

   // Add a camera, light, and material
   SoPerspectiveCamera* myCamera = new SoPerspectiveCamera;
   root->addChild(myCamera);
   root->addChild(new SoDirectionalLight);
   SoMaterial* myMaterial = new SoMaterial;
   myMaterial->diffuseColor.setValue( 1.0, 0.0, 0.0 );
   root->addChild(myMaterial);

   // Create a Text3 object, and connect to the realTime field
   SoText3* myText = new SoText3;
   root->addChild(myText);
   myText->string.connectFrom(SoDB::getGlobalField("realTime"));

   SoXtRenderArea* myRenderArea = new SoXtRenderArea(myWindow);
   myCamera->viewAll( root, myRenderArea->getSize() );
   myRenderArea->setSceneGraph(root);
   myRenderArea->setTitle("Date & Time");
   myRenderArea->show();

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

   return 0;
}
```

# 5 Engine per animazioni

I seguenti engine vengono usati per animare oggetti nel grafo della scena. Ognuno di questi engine ha un campo *timeIn* che è connesso automaticamente al global field **realTime** quando l'engine viene creato. Comunque sia questo campo può essere connesso a qualsiasi altra sorgente temporale.

- **SoElapsedTime**: funziona come un cronometro, l'output viene generato quando è passato un certo tempo fissato.

- **SoOneShot**: funziona per un certo periodo di tempo, poi si ferma.

- **SoTimeCounter**: cicla da un minimo ad un massimo ad una data frequenza.

**Elapsed-Time Engine**

| | |
|---|---|
| **SoElapsedTime** | |
| Include File | `#include <Inventor/engines/SoElapsedTime.h>` |
| Inputs from **SoElapsedTime** | **SoSFTime** *timeIn*<br>**SoSFFloat** *speed*<br>**SoSFBool** *on*<br>**SoSFBool** *pause*<br>**SoSFTrigger** *reset* |
| Outputs from **SoElapsedTime** | (**SoSFTime**) *timeOut* |

Questo tipo di engine rappresenta una sorgente di tempo controllabile. Si può avviare, fermare, mettere in pausa e controllare la velocità di questo engine come fosse un cronografo. Se si mette in pausa, questo engine finisce di aggiornare il campo *timeOut*, ma internamente prosegue a contare. Quando si disattiva la pausa, il campo *timeOut* viene aggiornato alla posizione corrente del counter.

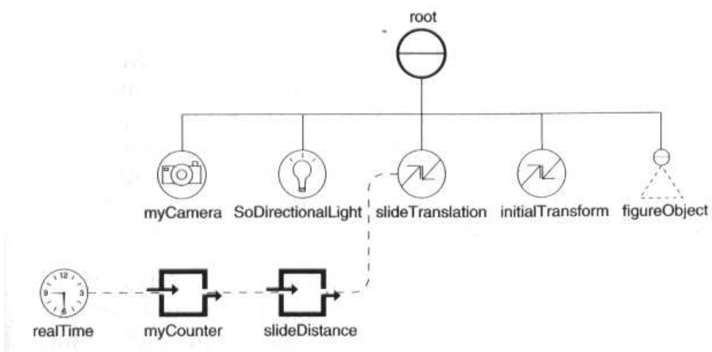


Figura 4: Grafo della scena dell'esempio con elapsed-time engine

L'esempio seguente, che ha come grafo quello di figura 4, usa l'output di un elapsed-time engine per controllare la traslazione di una figura. L'output *timeOut* dell'engine (*myCounter*) è connesso ad un engine di tipo **SoComposeVec3f** (*slideDistance*). Questo secondo engine inserisce il valore di *timeOut* all'interno della $x$ di un vettore. Una volta che il valore è in formato vettoriale, può essere connesso al campo *translation* del nodo *slideTranslation*.

```
// Set up transformations
SoTranslation* slideTranslation = new SoTranslation;
root->addChild(slideTranslation);
SoTransform* initialTransform = new SoTransform;
initialTransform->translation.setValue(-5., 0., 0.);
initialTransform->scaleFactor.setValue(10., 10., 10.);
initialTransform->rotation.setValue(SbVec3f(1,0,0), M_PI/2.);
root->addChild(initialTransform);

// Read the figure object from a file and add to the scene
SoInput myInput;
if(!myInput.openFile("jumpyMan.iv")) return 1;

SoSeparator* figureObject = SoDB::readAll(&myInput);
if(figureObject == 0) return 1;

root->addChild(figureObject);
```

```
// Make the X translation value change over time.
SoElapsedTime* myCounter = new SoElapsedTime;
SoComposeVec3f* slideDistance = new SoComposeVec3f;
slideDistance->x.connectFrom(&myCounter->timeOut);
slideTranslation->translation.connectFrom(&slideDistance->vector);
```

### One-Shot Engine

| **SoOneShot** | |
|---:|---|
| Include File | `#include <Inventor/engines/So.h>` |
| enum **Flags** | `SoOneShot::RETRIGGERABLE`<br>`SoOneShot::HOLD_FINAL` |
| Inputs from **SoOneShot** | **SoSFTime** *timeIn*<br>**SoSFTime** *duration*<br>**SoSFTrigger** *trigger*<br>**SoSFBitMask** *flags*<br>**SoSFBool** *disable* |
| Outputs from **SoOneShot** | (**SoSFTime**) *timeOut*<br>(**SoSFBool**) *isActive*<br>(**SoSFFloat**) *ramp* |

L'engine **SoOneShot**, quando il campo di input *trigger* viene modificato, si attiva per un tempo *duration* aggiornando il campo di output *timeOut*. Il campo *ramp* è 0.0 quando viene attivato l'engine, ed è 1.0 quando l'engine si disattiva dopo il tempo *duration*. Questo campo è molto utile ad esempio per essere connesso ad altri campi per interpolare trasformazioni geometriche.

L'engine ha due flag settabili nel campo *flags* di tipo **SoSFBitMask**.
`SoOneShot::RETRIGGERABLE` indica che può essere fatto ripartire anche nel mezzo di un suo ciclo di attivazione, altrimenti (il default) l'engine ignora la riattivazione fino a quando non conclude il ciclo.
`SoOneShot::HOLD_FINAL` specifica cosa può accadere alla fine del ciclo. Se questo flag non è settato (il default) tutti gli output ritornano a 0 quando il ciclo è terminato, altrimenti alla fine del ciclo il campo *isActive* ritorna a 0, ma i campi *ramp* e *timeOut* mantengono il proprio valore finale.

### Time-Counter Engine

| **SoTimeCounter** | |
|---:|---|
| Include File | `#include <Inventor/engines/SoTimeCounter.h>` |
| Inputs from **SoTimeCounter** | **SoSFTime** *timeIn*<br>**SoSFShort** *min*<br>**SoSFShort** *max*<br>**SoSFShort** *step*<br>**SoSFBool** *on*<br>**SoSFFloat** *frequency*<br>**SoSFFloat** *duty*<br>**SoSFShort** *reset*<br>**SoSFTrigger** *syncIn* |
| Outputs from **SoTimeCounter** | (**SoSFShort**) *output*<br>(**SoSFTrigger**) *syncOut* |

L'engine **SoTimeCounter** conta da un valore minimo *min* fino ad un valore massimo *max* con salti pari a *step* (default 1). Il campo *frequency* indica quanti cicli min-to-max devonno essere eseguiti al secondo. A differenza dei precedenti due engine, il **SoTimeCounter** non produce come output un tempo, ma produce come output il valore corrente del contatore. Ogni volta che entra in funzione, produce un "segnale" *syncOut* che permette di sincronizzare altri engine (ad esempio potrebbe essere usato come input nel campo *trigger* dell'engine One-Shot).

Il seguente esempio usa l'output di due engine **SoTimeCounter** per controllare il movimento orizzontale e verticale di una figura. Nell'esempio vengono us-

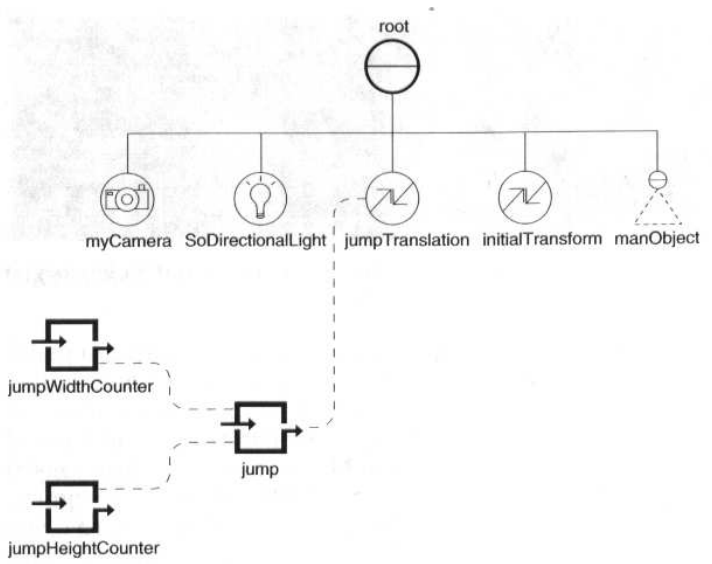


Figura 5: Grafo della scena dell'esempio con Time-Counter engine

ati 3 engine (si veda il grafo di figura 5. L'output dell'engine *jumpWidthCounter* (di tipo **SoTimeCounter**) viene connesso al campo $x$ dell'engine *jump* di tipo **SoComposeVec3f**. L'output dell'engine *jumpHeightCounter* (di tipo **SoTimeCounter**) viene connesso al campo $y$ dell'engine *jump*. L'engine *jump* crea il vettore ed inserisce tale vettore nel campo *translation* del nodo *jumpTranslation*. Il codice è visibile di seguito.

```
// Set up transformations
SoTranslation* jumpTranslation = new SoTranslation;
root->addChild(jumpTranslation);
SoTransform* initialTransform = new SoTransform;
initialTransform->translation.setValue( -20., 0., 0. );
initialTransform->scaleFactor.setValue( 40., 40., 40. );
initialTransform->rotation.setValue( SbVec3f(1,0,0), M_PI/2. );
root->addChild(initialTransform);

// Read the man object from a file and add to the scene
SoInput myInput;
if(!myInput.openFile("jumpyMan.iv")) return 1;
SoSeparator* manObject = SoDB::readAll(&myInput);
if(manObject == 0) return 1;

root->addChild(manObject);

// Create two counters, and connect to X and Y translations.
// The Y counter is small and high frequency.
// The X counter is large and low frequency.
// This results in small jumps across the screen,
```

```
// left to right, again and again and again and ....
SoTimeCounter* jumpHeightCounter = new SoTimeCounter;
SoTimeCounter* jumpWidthCounter = new SoTimeCounter;
SoComposeVec3f* jump = new SoComposeVec3f;

jumpHeightCounter->max = 4;
jumpHeightCounter->frequency = 1.5;
jumpWidthCounter->max = 40;
jumpWidthCounter->frequency = 0.15;

jump->x.connectFrom(&jumpWidthCounter->output);
jump->y.connectFrom(&jumpHeightCounter->output);
jumpTranslation->translation.connectFrom(&jump->vector);
```

# 6 Gate engine

| **SoGate** | |
|---|---|
| Include File | `#include <Inventor/engines/So.h>` |
| Inputs from **SoGate** | **SoSFBool** *enable*<br>**SoSFTrigger** *trigger*<br><**ioType**> *input* |
| Outputs from **SoGate** | (<**ioType**>) *output* |
| Methods from **SoGate** | `SoGate(SoType ioType)` |

Il gate engine permette di copiare selettivamente i valori di input sull'output. Quando si usa un gate engine, bisogna passare al costruttore il tipo dei campi di input ed output. Questo tipo deve essere un tipo per campi multivalore (MF). Se usiamo valori singoli, bisognerà usare il corrispondente tipo multivalore, poiché Inventor eseguirà la conversione automaticamente. Se il campo *enable* è TRUE, i dati in input sono copiati sull'output dell'engine ogni volta che vengono ricevuti. Se invece il campo è FALSE, allora i dati vengono copiati sull'output solo quando viene mandato un "segnale" al campo *trigger* (come se si procedesse ad un campionamento a tenuta). Il campo *trigger* può essere "toccato" con il metodo `touch()` o con il metodo `setValue()`.

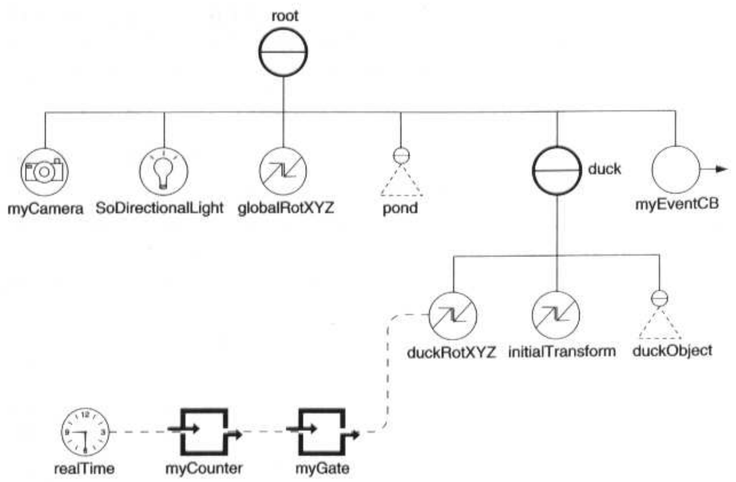


Figura 6: Grafo della scena dell'esempio con gate engine

L'esempio seguente (di cui il grafo può essere osservato in figura 6) connette un engine Elapsed-Time (*myCounter*) ad un gate engine (*myGate*). Premendo il bottone del mouse si abilita o disabilita il gate engine, il quale controlla il movimento di un oggetto sullo schermo.

```
   // Duck group
   SoSeparator* duck = new SoSeparator;
   root->addChild(duck);

   // Read the duck object from a file and add to the group
   SoInput myInput;

   if(!myInput.openFile("duck.iv")) return 1;

   SoSeparator* duckObject = SoDB::readAll(&myInput);
   if(duckObject == 0) return 1;

   // Set up the duck transformations
   SoRotationXYZ* duckRotXYZ = new SoRotationXYZ;
   duck->addChild(duckRotXYZ);
   SoTransform* initialTransform = new SoTransform;
   initialTransform->translation.setValue( 0., 0., 3. );
   initialTransform->scaleFactor.setValue( 6., 6., 6. );
   duck->addChild(initialTransform);
   duck->addChild(duckObject);

   // Update the rotation value if the gate is enabled.
   SoGate* myGate = new SoGate(SoMFFloat::getClassTypeId());
   SoElapsedTime* myCounter = new SoElapsedTime;
   myGate->input->connectFrom(&myCounter->timeOut);
   duckRotXYZ->axis = SoRotationXYZ::Y;  // rotate about Y axis
   duckRotXYZ->angle.connectFrom(myGate->output);

   // Add an event callback to catch mouse button presses.
   // Each button press will enable or disable the duck motion.
   SoEventCallback* myEventCB = new SoEventCallback;
   myEventCB->addEventCallback( SoMouseButtonEvent::getClassTypeId(),
              myMousePressCB, myGate);
   root->addChild(myEventCB);

...

// This routine is called for every mouse button event.
void myMousePressCB( void* userData, SoEventCallback* eventCB )
{
   SoGate* gate = (SoGate*)userData;
   const SoEvent* event = eventCB->getEvent();

   // Check for mouse button being pressed
   if(SO_MOUSE_PRESS_EVENT(event, ANY)) {

      // Toggle the gate that controls the duck motion
      if (gate->enable.getValue())
         gate->enable.setValue(FALSE);
      else
         gate->enable.setValue(TRUE);
      eventCB->setHandled();
   }
}
```

# 7 Engine aritmetici

Per convenzione, tutti gli input e gli output degli engine aritmetici hanno campi multivalore. Se l'engine ha più di un input ed uno di questi è un array, allora anche l'output sarà un array di valori multivalore.

### Engine Booleano

| **SoBoolOperation** | |
|---|---|
| Include File | `#include <Inventor/engines/SoBoolOperation.h>` |
| enum **Operation** | `SoBoolOperation::CLEAR`<br>`SoBoolOperation::SET`<br>`SoBoolOperation::A)`<br>`SoBoolOperation::NOT_A`<br>`SoBoolOperation::B`<br>`SoBoolOperation::NOT_B`<br>`SoBoolOperation::A_OR_B`<br>`SoBoolOperation::NOT_A_OR_B`<br>`SoBoolOperation::A_OR_NOT_B`<br>`SoBoolOperation::NOT_A_OR_NOT_B`<br>`SoBoolOperation::A_AND_B`<br>`SoBoolOperation::NOT_A_AND_B`<br>`SoBoolOperation::A_AND_NOT_B`<br>`SoBoolOperation::NOT_A_AND_NOT_B`<br>`SoBoolOperation::A_EQUALS_B`<br>`SoBoolOperation::A_NOT_EQUALS_B` |
| Inputs from **SoBoolOperation** | **SoMFBool** *a*<br>**SoMFBool** *b*<br>**SoMFEnum** *operation* |
| Outputs from **SoBoolOperation** | (**SoMFBool**) *output*<br>(**SoMFBool**) *inverse* |

L'engine booleano ha in input due valori booleani, *a* e *b* ed un valore di tipo **SoSFEnum** che rappresenta l'operazione (*operation*) da effettuare sugli input booleani. Le operazioni ed il significato sono riportate in seguito.

| **Operation** | **Output is TRUE if** |
|---|---|
| `SoBoolOperation::CLEAR` | never TRUE |
| `SoBoolOperation::SET` | always TRUE |
| `SoBoolOperation::A` | A is TRUE |
| `SoBoolOperation::NOT_A` | A is FALSE |
| `SoBoolOperation::B` | B is TRUE |
| `SoBoolOperation::NOT_B` | B is FALSE |
| `SoBoolOperation::A_OR_B` | A is TRUE or B is TRUE |
| `SoBoolOperation::NOT_A_OR_B` | A is FALSE or B is TRUE |
| `SoBoolOperation::A_OR_NOT_B` | A is TRUE or B is FALSE |
| `SoBoolOperation::NOT_A_OR_NOT_B` | A is FALSE or B is FALSE |
| `SoBoolOperation::A_AND_B` | A is TRUE and B is TRUE |
| `SoBoolOperation::NOT_A_AND_B` | A is FALSE and B is TRUE |
| `SoBoolOperation::A_AND_NOT_B` | A is TRUE and B is FALSE |
| `SoBoolOperation::NOT_A_AND_NOT_B` | A is FALSE and B is FALSE |
| `SoBoolOperation::A_EQUALS_B` | A equals B |
| `SoBoolOperation::A_NOT_EQUALS_B` | A does not equal B |

L'engine ha poi due output, *output* ed *inverse*. Il campo *inverse* é TRUE se *output* é FALSE e viceversa. Ovviamente l'engine funziona anche su array di booleani in ingresso e uscita.

L'esempio successivo modifica il precedente inserendo un engine booleano che effettua il movimento di una papera piccola rispetto il movimento della papera grande. La papera piccola si muove quando la papera grande viene fermata allo schiacciare del bottone del mouse.

```
// Bigger duck group
SoSeparator* bigDuck = new SoSeparator;
root->addChild(bigDuck);
SoRotationXYZ* bigDuckRotXYZ = new SoRotationXYZ;
bigDuck->addChild(bigDuckRotXYZ);
SoTransform* bigInitialTransform = new SoTransform;
bigInitialTransform->translation.setValue(0., 0., 3.5);
bigInitialTransform->scaleFactor.setValue(6., 6., 6.);
bigDuck->addChild(bigInitialTransform);
bigDuck->addChild(duckObject);

// Smaller duck group
SoSeparator* smallDuck = new SoSeparator;
root->addChild(smallDuck);
SoRotationXYZ* smallDuckRotXYZ = new SoRotationXYZ;
smallDuck->addChild(smallDuckRotXYZ);
SoTransform* smallInitialTransform = new SoTransform;
smallInitialTransform->translation.setValue(0., -2.24, 1.5);
smallInitialTransform->scaleFactor.setValue(4., 4., 4.);
smallDuck->addChild(smallInitialTransform);
smallDuck->addChild(duckObject);

// Use a gate engine to start/stop the rotation of
// the bigger duck.
SoGate* bigDuckGate = new SoGate(SoMFFloat::getClassTypeId());
SoElapsedTime* bigDuckTime = new SoElapsedTime;
bigDuckGate->input->connectFrom(&bigDuckTime->timeOut);
bigDuckRotXYZ->axis = SoRotationXYZ::Y;  // Y axis
bigDuckRotXYZ->angle.connectFrom(bigDuckGate->output);

// Each mouse button press will enable/disable the gate
// controlling the bigger duck.
SoEventCallback* myEventCB = new SoEventCallback;
myEventCB->addEventCallback(
         SoMouseButtonEvent::getClassTypeId(),
         myMousePressCB, bigDuckGate);
root->addChild(myEventCB);

// Use a Boolean engine to make the rotation of the smaller
// duck depend on the bigger duck.  The smaller duck moves
// only when the bigger duck is still.
SoBoolOperation* myBoolean = new SoBoolOperation;
myBoolean->a.connectFrom(&bigDuckGate->enable);
myBoolean->operation = SoBoolOperation::NOT_A;

SoGate* smallDuckGate = new SoGate(SoMFFloat::getClassTypeId());
SoElapsedTime* smallDuckTime = new SoElapsedTime;
smallDuckGate->input->connectFrom(&smallDuckTime->timeOut);
smallDuckGate->enable.connectFrom(&myBoolean->output);
smallDuckRotXYZ->axis = SoRotationXYZ::Y;  // Y axis
smallDuckRotXYZ->angle.connectFrom(smallDuckGate->output);
```

**Engine Calcolatore**

| SoCalculator | |
|---|---|
| Include File | `#include <Inventor/engines/SoCalculator.h>` |
| Inputs from **SoCalculator** | |
| **SoMFFloat** | *a, b, c, d, e, f, g, h* |
| **SoMFVec3f** | *A, B, C, D, E, F, G, H* |
| **SoMFString** | expression |
| Outputs from **SoCalculator** | |
| **SoEngineOutput** | *oa, ob, oc, od* (**SoMFFloat**) |
| **SoEngineOutput** | *oA, oB, oC, oD* (**SoMFVec3f**) |

L'engine calcolatore ha svariati input (come visibile nella tabella di sopra) e svariati output. L'espressione o le espressioni inserite nell'engine permettono di calcolare i valori di output dai valori di input. Le stringhe inserite devono avere la forma:

$$lhs = rhs.$$

L'elemento a sinistra ($lhs$) puó essere una variabile di uscita oppure una variabile temporanea (indicate con $ta \dots th$ e con $tA \dots tH$). L'espressione a destra puó supportare i seguenti elementi ed operatori:

| **Type of Operator** | **Example** |
|---|---|
| Binary operators | $+ - * / < > >= <= == != \&\& \|\|$ |
| Unary operators | - ! |
| Ternary operator | *cond ? trtteexpr : falseexpr* |
| Parentheses | (expr) |
| Vector indexing | vec[int] |
| Functions | func( expr, ... ) |
| Terms | integer or floating-point constants |
| | named constants such as MAXFLOAT, MINFLOAT, M_LOG2E, M_PI |
| | the names of the calculator engine's inputs, outputs, and temporary variables |
| | (a, b, A, B, oa, ob, ta, tb, tA, tB, and so on) |

Un semplice uso di questo engine è il seguente:

| **Input** | **Output** |
|---|---|
| 2 vettori (A, B) | oA ($-f * prod_vett(A, B)$) |
| 2 scalari (a, f) | oa (conversione di a in radianti) |

Per specificare le espressioni nell'engine *calc* il codice è:

```
calc->expression.set1Value( 0, "oa = a * M_PI / 180" );
calc->expression.set1Value( 1, "oA = -f * cross(A,B) );
```

L'esempio seguente mostra come utilizzare un engine calcolatore per muovere un fiore lungo un cammino. Il grafo della scena, insieme ad una schematizzazione dell'engine calcolatore, è visibile in figura 7. L'engine calcola una curva chiusa e planare. L'output dell'engine è connesso alla traslazione applicata al fiore che quindi si muoverà lungo la curva calcolata.

```
// Flower group
SoSeparator* flowerGroup = new SoSeparator;
root->addChild(flowerGroup);

// Read the flower object from a file and add to the group
if (!myInput.openFile("flower.iv")) return 1;
```

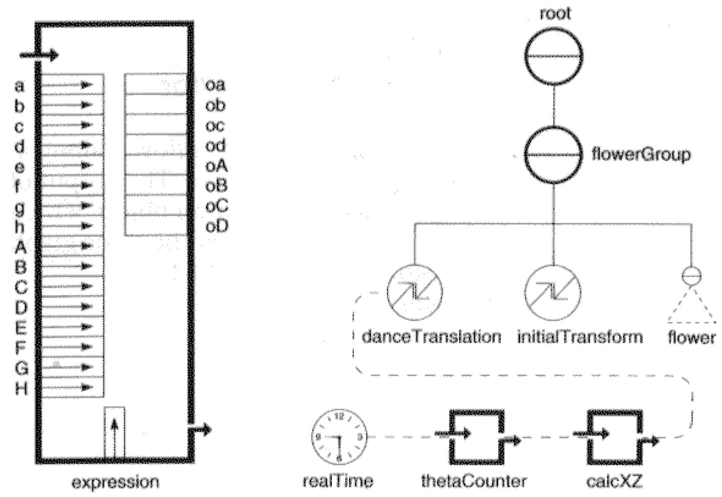


Figura 7: Schematizzazione di un engine calcolatore e grafo della scena dell'esempio d'uso di tale engine

```
SoSeparator* flower = SoDB::readAll(&myInput);
if (flower == 0) return 1;

// Set up the flower transformations
SoTranslation* danceTranslation = new SoTranslation;
SoTransform* initialTransform = new SoTransform;
flowerGroup->addChild(danceTranslation);
initialTransform->scaleFactor.setValue(10., 10., 10.);
initialTransform->translation.setValue(0., 0., 5.);
flowerGroup->addChild(initialTransform);
flowerGroup->addChild(flower);

// Set up an engine to calculate the motion path:
// r = 5*cos(5*theta); x = r*cos(theta); z = r*sin(theta)
// Theta is incremented using a time counter engine,
// and converted to radians using an expression in
// the calculator engine.
SoCalculator* calcXZ = new SoCalculator;
SoTimeCounter* thetaCounter = new SoTimeCounter;

thetaCounter->max = 360;
thetaCounter->step = 4;
thetaCounter->frequency = 0.075;

calcXZ->a.connectFrom(&thetaCounter->output);
calcXZ->expression.set1Value(0, "ta=a*M_PI/180");   // theta
calcXZ->expression.set1Value(1, "tb=5*cos(5*ta)");  // r
calcXZ->expression.set1Value(2, "td=tb*cos(ta)");   // x
calcXZ->expression.set1Value(3, "te=tb*sin(ta)");   // z
calcXZ->expression.set1Value(4, "oA=vec3f(td,0,te)");
danceTranslation->translation.connectFrom(&calcXZ->oA);
```

## 8 Nodi particolari per animazioni

Gli engine si devono connettere a campi di altri nodi. Inventor mette a disposizione alcuni nodi che hanno un built-in engine e sono automaticamente connessi ad esso. Questi nodi sono:

- **SoRotor**, un nodo di trasformazioni che cambia l'angolo di rotazione mantenendo costante l'asse.
- **SoPendulum**, anch'esso un nodo trasformazione che oscilla tra due differenti rotazioni.
- **SoShuttle**, nodo di trasformazione che oscilla tra due traslazioni.
- **SoBlinker**, nodo che cicla sui suoi figli attivandone uno alla volta.

**SoRotor** Questo nodo è derivato dal nodo **SoRotation** e cambia l'angolo di rotazione ad una certa velocità impostabile. Il nodo ha tre campi:

- *rotation* in cui si specifica la rotazione, quindi l'asse e l'angolo iniziale;
- *speed* in cui si specifica il numero di cicli per secondo;
- *on* che sarà impostato a TRUE per attivare l'engine built-in (default).

L'esempio seguente illustra l'uso del nodo per far muovere le pale di un mulino a vento.

```
SoSeparator* readFile(const char* filename)
{
   // Open the input file
   SoInput mySceneInput;
   if (!mySceneInput.openFile(filename)) {
      fprintf(stderr, "Cannot open file %s\n", filename);
      return 0;
   }

   // Read the whole file into the database
   SoSeparator* myGraph = SoDB::readAll(&mySceneInput);
   if (myGraph == 0) {
      fprintf(stderr, "Problem reading file\n");
      return 0;
   }

   mySceneInput.closeFile();
   return myGraph;
}

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

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

   // Read in the data for the windmill tower

   SoSeparator *windmillTower = readFile("windmillTower.iv");
```

```
    root->addChild(windmillTower);

    // Add a rotor node to spin the vanes
    SoRotor* myRotor = new SoRotor;
    myRotor->rotation.setValue(SbVec3f(0, 0, 1), 0);  // z axis
    myRotor->speed = 0.2;
    root->addChild(myRotor);

    // Read in the data for the windmill vanes

    SoSeparator* windmillVanes = readFile("windmillVanes.iv");

    root->addChild(windmillVanes);

    // Create a viewer
    SoXtExaminerViewer* myViewer = new SoXtExaminerViewer(myWindow);

    // attach and show viewer
    myViewer->setSceneGraph(root);
    myViewer->setTitle("Windmill");
    myViewer->show();

    // Loop forever
    SoXt::show(myWindow);
    SoXt::mainLoop();
}
```

**SoBlinker** Il nodo deriva da SoSwitch e cicla tra i suoi figli cambiando il valore del campo *whichChild*. Esso ha i seguenti campi:

- *whichChild* che indica l'indice del figlio attualmente interessato al traversal;
- *speed* in cui si specifica il numero di cicli per secondo;
- *on* che sarà impostato a TRUE per attivare l'engine built-in (default).

Se il nodo ha un solo figlio, esso ciclerà tra questo e `SO_SWITCH_NODE`.

L'esempio che segue visualizza un testo accendendolo e spegnendolo.

```
    // Add the non-blinking part of the sign to the root
    root->addChild(eatAt);

    // Add the fast-blinking part to a blinker node
    SoBlinker* fastBlinker = new SoBlinker;
    root->addChild(fastBlinker);
    fastBlinker->speed = 2;        // blinks 2 times a second
    fastBlinker->addChild(josie);

    // Add the slow-blinking part to another blinker node
    SoBlinker* slowBlinker = new SoBlinker;
    root->addChild(slowBlinker);
    slowBlinker->speed = 0.5;     // 2 secs per cycle; 1 on, 1 off
    slowBlinker->addChild(frame);
```