Open Inventor
4 - Viewing and Light

Claudio Baldazzi *Maurizio Patrignani*

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

23 aprile 1998

1 Introduzione

In questi appunti si parlerà di due classi di nodi che influenzano la visualizzazione di una scena: *cameras* e *lights*. In Inventor, come nel mondo reale, le luci servono ad illuminare gli ogetti della scena. Se il grafo della scena non contiene luci e si sta usando il modello di illuminazione di default (Phong), gli oggetti non sono visibili. La *camera* di Inventor è come se fosse l'obiettivo di una telecamera sulla scena, un pó come fosse il nostro occhio che osserva. In pratica serve per creare una fotografia bidimensionale della scena. 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 Cameras

Un nodo *camera* genera una proiezione bidimensionale di tutti i nodi che si trovano dopo di esso durante il traversal del grafo. Generalmente, quindi, si pone un nodo camera in alto a sinistra del grafo della scena, in modo da poter visualizzare tutti gli oggetti creati. Ogni grafo di scena puó contenere solo una camera attiva, cioé vi possono essere anche piú nodi camera, ma solo uno sarà attivo ogni rendering action. I nodi camera derivano tutti dalla classe astratta **SoCamera**. Essendo questa una classe astratta, non potremo mai instanziare oggetti di questa classe, ma solo delle classi derivate, cioé **SoOrthographicCamera** e **SoPerspectiveCamera**.

I campi della classe **SoCamera** (quindi comuni ad entrambe le classi derivate) sono:

- **viewportMapping**: importante quando l'*aspectRatio* della camera é differente dall'*aspectRatio* del viewport. Vedremo piú avanti il suo uso.
- **position**: punto in cui è posizionato l'obbiettivo (risente delle trasformazioni geometriche).

- **orientation**: direzione in cui è orientata la camera (risente delle trasformazioni geometriche).
- **aspectRatio**: rapporto larghezza-altezza della camera. Costanti predefinite sono:
  - `SO_ASPECT_SQUARE` (1/1)
  - `SO_ASPECT_VIDEO` (4/3)
  - `SO_ASPECT_HDTV` (16/9)
- **nearDistance**: distanza tra la posizione della camera e il piú vicino piano di clipping.
- **farDistance**: distanza tra la posizione della camera e il piú lontano piano di clipping.
- **focalDistance**: distanza tra la posizione della camera e il punto di messa a fuoco. è usato dall'examiner viewer.

Dinamicamente si puó variare l'orientazione della camera con il metodo **pointAt()** il cui prototipo di funzione è il seguente:

```
void    pointAt( const SbVec3f& targetPoint )
```

Il metodo vuole come parametro un punto a cui orientarsi. Se possibile Inventor tenterà di orientare la camera parallelamente all'asse y o all'asse z. Altri due metodi importanti di questa classe sono **viewAll()** e **getViewVolume()**. Il metodo **viewAll()** permette di settare la camera in modo tale da visualizzare l'intero grafo di scena usando l'orientazione corrente. La sintassi è la seguente:

```
void    viewAll( SoNode* root, const SbViewportRegion& vp,
                 float slack=1.0 )
```

in cui *root* è la root del grafo di scena, *vp* è il viewport corrente e *slack* serve per posizionare i piani di clipping in un certo modo. Il metodo **getViewVolume()** ritorna il volume di vista e, generalmente, é usato in relazione ad un *picking*.

2.1 Orthographic Camera

La camera appartenente alla classe **SoOrthographicCamera** produce proiezioni parallele con nessuna distorsione a causa della distanza. Un nodo di questo tipo ha un altro campo oltre a quelli già definiti:

- **height**: specifica l'altezza del volume di vista.

Il volume di vista è un parallelepipedo, visibile in figura 1 con i campi dell'oggetto da riempire. L'altezza e l'aspect ratio determinano la larghezza della finestra:

$$width = height * aspectRatio.$$

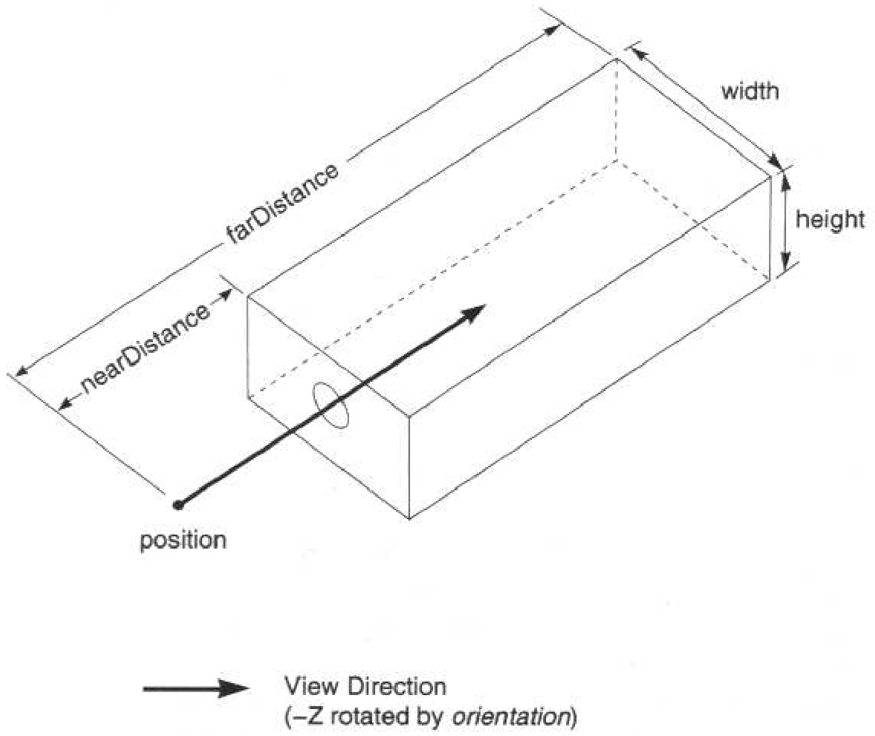

Figura 1: Volume di vista e proiezione di vista per un nodo **SoOrthographic-Camera**

2.2 Perspective Camera

La camera appartenente alla classe **SoPerspectiveCamera** produce proiezioni prospettiche con distorsione a causa della distanza. Tale proiezione emula l'occhio umano. Un nodo di questo tipo ha un altro campo oltre a quelli già definiti:

- **heightAngle**: specifica l'angolo verticale in radianti del volume di vista.

Il volume di vista è un tronco di cono, visibile in figura 2 con i campi dell'oggetto da riempire. L'angolo verticale e l'aspect ratio determinano l'angolo orizzontale della finestra:

$$widthAngle = heightAngle * aspectRatio.$$

2.3 Rendering Action and Projection

Durante una Rendering Action, se viene incontrato un nodo camera attivo, viene modifica la matrice di vista poi usata per proiettare sullo schermo la scena. Fisicamente è come se venissero effettuati i seguenti passi (ognuno modificherà parte della matrice):

1. La camera viene posizionata sulla scena.

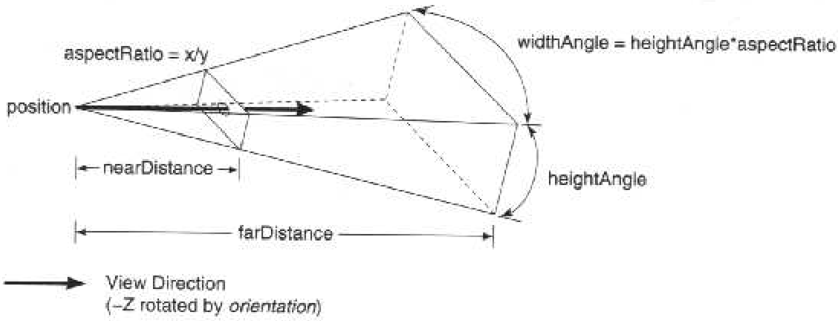

Figura 2: Volume di vista e proiezione di vista per un nodo **SoPerspective-Camera**

2. La camera crea il volume di vista, basandosi sui piani di clipping, sull'aspect ratio, sull'altezza o l'angolo di altezza (a seconda del tipo di camera). Gli oggetti al di fuori del volume di vista vengono eliminati (clipping).
3. Proiezione del volume di vista in una immagine bidimensionale. In questo momento esiste una matrice che mi permette, in maniera semplice, di mappare gli oggetti tridimensionali nella finestra bidimensionale che apparirà sullo schermo.
4. Il resto della scena viene trattato usando la proiezione creata dalla camera.

2.4 Window and Viewport

Un *viewport* è un'area rettangolare in cui la scena è renderizzata. Di default, il viewport ha le stesse dimensioni della finestra (**SoXtRenderArea**).

Il campo **viewportMapping** della classe **SoCamera** serve a specificare come effettuare il mapping della proiezione dalla camera al viewport quando l'*aspectRatio* della camera è differente dall'*aspectRatio* del viewport. Tre sono le scelte possibili contenere tutta la window della camera nel viewport mantenendo inalterato l'aspect ratio:

- `CROP_VIEWPORT_FILL_FRAME`: modifica il viewport in modo da contenere la window della camera. Lo spazio del viewport non usato viene riempito di grigio.
- `CROP_VIEWPORT_LINE_FRAME`: modifica il viewport in modo da contenere la window della camera. Il bordo del viewport viene disegnato con una linea.
- `CROP_VIEWPORT_NO_FRAME`: modifica il viewport in modo da contenere la window della camera. Lo spazio del viewport non usato non viene indicato.

Due invece sono le scelte possibili per modificare la window della camera in modo da essere disegnata completamente nel viewport:

- `ADJUST_CAMERA`: modifica la window generata dalla camera in modo da essere disegnata tutta sul viewport. L'immagine non viene distorta, quindi l'aspect ratio viene mantenuto. Eventualmente la window viene tagliata ai bordi.

- LEAVE_ALONE: modifica la window generata dalla camera in modo da essere disegnata tutta sul viewport. L'immagine viene distorta, quindi l'aspect ratio non viene mantenuto. La window viene immersa nel viewport.

Si veda figura 3 per degli esempi di mapping window-viewport.

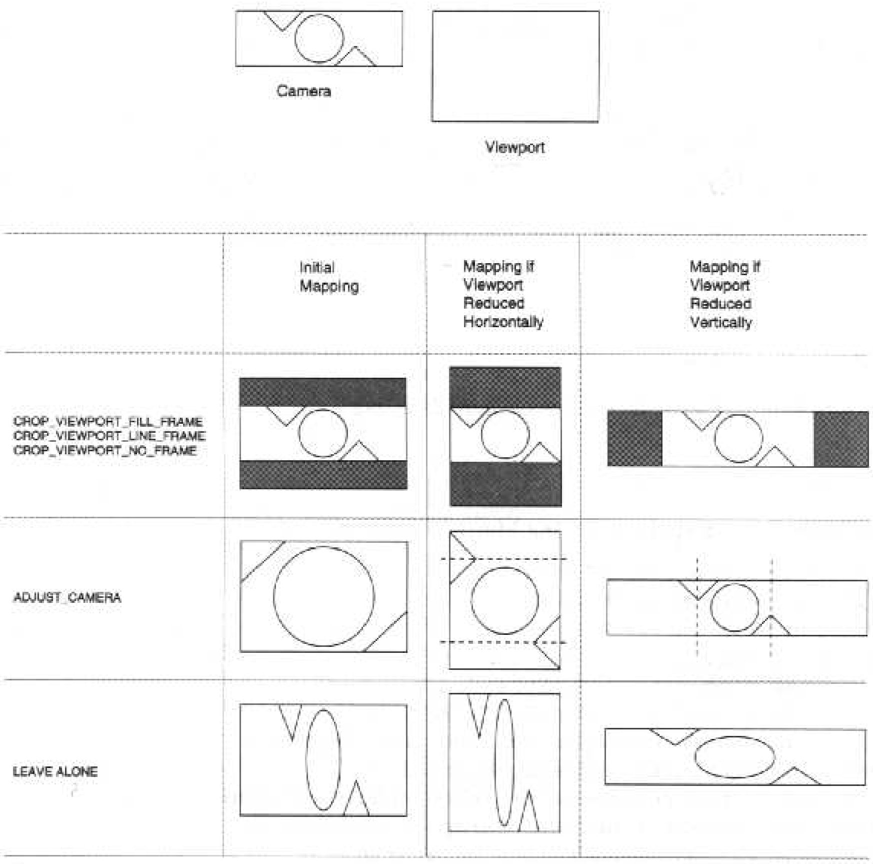

Figura 3: Come l'Aspect Ratio della Camera viene *mappato* sulla Viewport.

```
/*--------------------------------------------------------------
 *  This is an example from the Inventor Mentor,
 *  chapter 4, example 1.
 *
 *  Camera example.
 *  A blinker node is used to switch between three
 *  different views of the same scene. The cameras are
 *  switched once per second.
 *------------------------------------------------------------*/

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

```
#include <Inventor/SoDB.h>
#include <Inventor/SoInput.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/nodes/SoBlinker.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoOrthographicCamera.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoTransform.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 a blinker node and put it in the scene. A blinker
   // switches between its children at timed intervals.
   SoBlinker* myBlinker = new SoBlinker;
   root->addChild(myBlinker);

   // Create three cameras. Their positions will be set later.
   // This is because the viewAll method depends on the size
   // of the render area, which has not been created yet.
   SoOrthographicCamera* orthoViewAll =new SoOrthographicCamera;
   SoPerspectiveCamera*  perspViewAll = new SoPerspectiveCamera;
   SoPerspectiveCamera*  perspOffCenter =new SoPerspectiveCamera;
   myBlinker->addChild(orthoViewAll);
   myBlinker->addChild(perspViewAll);
   myBlinker->addChild(perspOffCenter);

   // Create a light
   root->addChild(new SoDirectionalLight);

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

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

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

   SoMaterial* myMaterial = new SoMaterial;
   myMaterial->diffuseColor.setValue(0.8, 0.23, 0.03);
   root->addChild(myMaterial);
   root->addChild(fileContents);

   SoXtRenderArea* myRenderArea = new SoXtRenderArea(myWindow);

   // Establish camera positions.
   // First do a viewAll on all three cameras.
   // Then modify the position of the off-center camera.
   SbViewportRegion myRegion(myRenderArea->getSize());
   orthoViewAll->viewAll(root, myRegion);
   perspViewAll->viewAll(root, myRegion);
   perspOffCenter->viewAll(root, myRegion);
```

```
    SbVec3f initialPos;
    initialPos = perspOffCenter->position.getValue();
    float x, y, z;
    initialPos.getValue(x,y,z);
    perspOffCenter->position.setValue(x+x/2., y+y/2., z+z/4.);

    myRenderArea->setSceneGraph(root);
    myRenderArea->setTitle("Cameras");
    myRenderArea->show();

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

    return 0;

}
```

3 Lights

Con il modello di illuminazione di default (Phong), il grafo della scena necessita almeno di una luce per essere visibile. La posizione del nodo luce nel grafo della scena determina:

- *cosa viene illuminato*: riceve la luce tutto ciò che segue il nodo che la definisce. Le condizioni di luce vanno a far parte dello stato del sistema durante il traversal del grafo della scena. Con dei nodi `SoSeparator` opportuni si può circoscrivere l'effetto di una sorgente luminosa.

- *dove è sita la luce*: alcuni campi del nodo luce definiscono un punto dello spazio in cui la luce risiede e/o la direzione dei raggi luminosi. Questi valori sono affetti dalla trasformazione geometrica attuale dello stato.

Ovviamente, aggiungere luci su luci ad una scena ha l'effetto di cumulare le loro intensità luminose fino a rendere tutti gli oggetti ugualmente illuminati.

Come i nodi camera derivano tutti dalla classe astratta `SoCamera`, così i nodi luce derivano dalla classe astratta `SoLight`. Questa classe non aggiunge metodi alla `SoNode`, ma si limita ad aggiungere tre campi:

- **on** (`SoSFBool`) specifica se la luce è accesa o spenta.

- **intensity** (`SoSFFloat`) specifica il valore dell'intensità della fonte luminosa in un range che va da 0.0 (luce spenta) a 1.0 (massima intensità).

- **color** (`SoSFColor`) specifica il colore della fonte luminosa.

La classe astratta `SoLight` ha a sua volta tre sottoclassi: `SoPointLight`, `SoDirectionalLight` e `SoSpotLight`. La figura 4 vuole rappresentare i raggi luminosi dei quattro diversi tipi di luce.

Diciamo subito che, dal punto di vista dell'efficienza, il rendering di una scena è più veloce se questa contiene luci direzionali che luci puntiformi, mentre è consistentemente più lento se ci sono luci di tipo spotlight.

3.1 `SoPointLight`

Si tratta di una luce che irradia in tutte le direzioni a partire da una sorgente puntiforme. Ha un solo campo aggiunto rispetto alla `SoLight`

- **location** (`SoSFVec3f`) specifica la posizione della sorgente nello spazio.

3.2 `SoDirectionalLight`

Si tratta di una luce direzionale che illumina uniformemente lungo una particolare direzione. L'effetto è quello di una sorgente puntiforme a distanza infinita. Ha un solo campo aggiunto rispetto alla classe `SoLight`:

- **direction** (`SoSFVec3f`) specifica la direzione dei raggi luminosi. (Attenzione: questi valori sono affetti dalla trasformazione geometrica corrente dello stato).

Tutti i raggi di una luce direzionale sono paralleli, e sarebbero riflessi nello stesso modo da tutti i punti di un poligono piatto, dando luogo ad una illuminazione piatta e di uguale intensità. Al contrario, l'intensità della luce proveniente da una `SoPointLight` su una superficie piatta varierebbe da punto a punto come varia l'angolo tra le normale alla superficie e il raggio incidente.

3.3 `SoSpotLight`

Questo tipo di luce riproduce l'effetto di una luce da palcoscenico: un cono di luce emessa da un punto verso una direzione primaria. Ha quattro campi addizionali rispetto alla classe `SoLight` (si veda la figura 5):

- **location** (`SoSFVec3f`) specifica la posizione della sorgente nello spazio.
- **direction** (`SoSFVec3f`) specifica la direzione primaria dei raggi luminosi. (Attenzione: anche questi valori sono affetti dalla trasformazione geometrica corrente dello stato).
- **dropOffRate** (`SoSFFloat`) specifica il tasso con il quale l'intensità luminosa decresce allontanandosi dalla direzione primaria: 0.0 corrisponde ad una intensità costante, sia al centro del cono di luce che alla periferia; 1.0 è il massimo valore di decremento luminoso.
- **cutOffAngle** (`SoSFFloat`) specifica (in radianti) l'angolo al di fuori del quale l'intensità luminosa sarà nulla. Questo angolo è misurato da un estremo del cono all'altro, e corrisponde dunque al doppio dell'angolo misurato dall'asse del cono ad una generatrice.

3.4 Un esempio con luci diverse

Nell'esempio che segue una scena presenta due sorgenti di luce: una luce direzionale stazionaria rossa ed una luce verde puntiforme che si muove avanti e indietro su un nodo `SoShuttle` (un nodo trasformazione che oscilla tra le due traslazioni assegnategli).

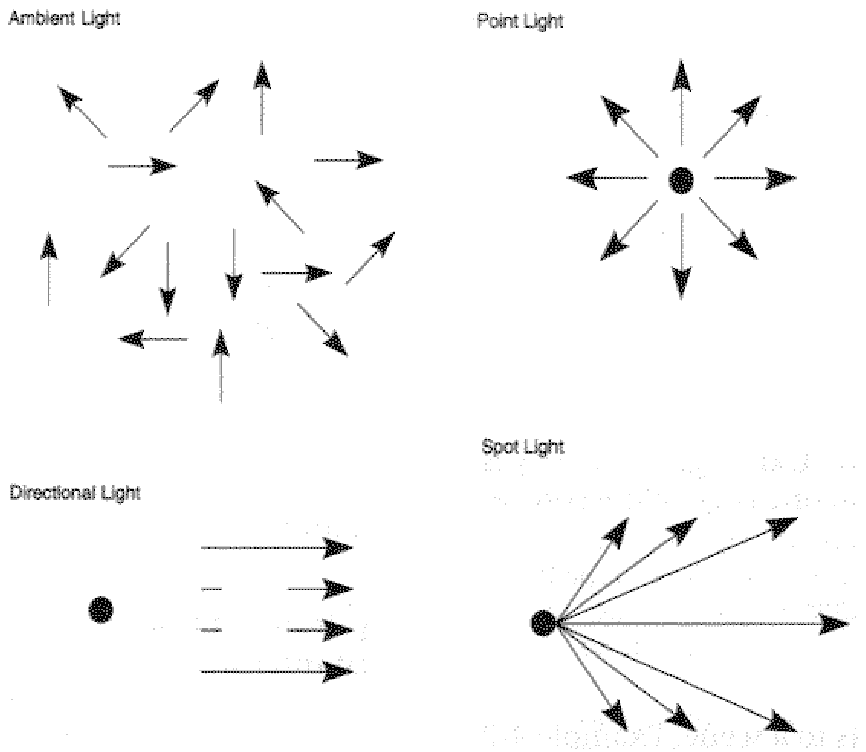

Figura 4: I vari tipi di luce definibili in Open Inventor. La luce ambiente può essere specificata tramite il campo `ambientIntensity` di un nodo `SoEnvironment`.

```
/*--------------------------------------------------------------
 *  This is an example from the Inventor Mentor,
 *  chapter 4, example 2.
 *
 *  Lights example.
 *  Read in an object from a file.
 *  Use the ExaminerViewer to view it with two light sources.
 *  The red directional light doesn't move; the green point
 *  light is moved back and forth using a shuttle node.
 *------------------------------------------------------------*/

#include <Inventor/SoDB.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPointLight.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoShuttle.h>
#include <Inventor/nodes/SoTransformSeparator.h>

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

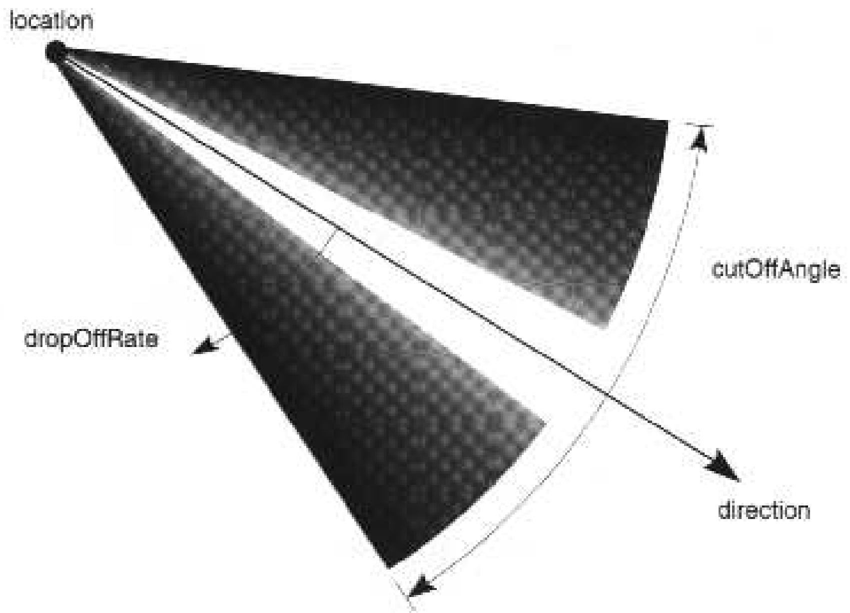

Figura 5: Una `SoSpotLight` e i relativi campi. Si noti che il `cutOffAngle` viene misurato da un estremo del cono di luce all'altro.

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

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

// Add a directional light
SoDirectionalLight* myDirLight = new SoDirectionalLight;
myDirLight->direction.setValue(0, -1, -1);
myDirLight->color.setValue(1, 0, 0);
root->addChild(myDirLight);

// Put the shuttle and the light below a transform separator.
// A transform separator pushes and pops the transformation
// just like a separator node, but other aspects of the state
// are not pushed and popped. So the shuttle's translation
// will affect only the light. But the light will shine on
// the rest of the scene.
SoTransformSeparator* myTransformSeparator =
    new SoTransformSeparator;
root->addChild(myTransformSeparator);

// A shuttle node translates back and forth between the two
// fields translation0 and translation1.
// This moves the light.
SoShuttle* myShuttle = new SoShuttle;
myTransformSeparator->addChild(myShuttle);
myShuttle->translation0.setValue(-2, -1, 3);
myShuttle->translation1.setValue( 1,  2, -3);

// Add the point light below the transformSeparator
```

```
   SoPointLight* myPointLight = new SoPointLight;
   myTransformSeparator->addChild(myPointLight);
   myPointLight->color.setValue(0, 1, 0);

   root->addChild(new SoCone);

   SoXtExaminerViewer* myViewer =
           new SoXtExaminerViewer(myWindow);
   myViewer->setSceneGraph(root);
   myViewer->setTitle("Lights");
   myViewer->setHeadlight(FALSE);
   myViewer->show();

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

   return 0;
}
```

4 LAB: Test Light

Nel seguito è fornito il listato di un programma per la visualizzazione di uno stesso oggetto con i vari tipi di illuminazione possibili.

Si provi a compilarlo e a vederne l'effetto.

```
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/SbLinear.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoRotation.h>
#include <Inventor/nodes/SoTranslation.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>

#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoPointLight.h>
#include <Inventor/nodes/SoSpotLight.h>

#include <math.h>

// Function Prototyping
SoSeparator* mkwall();
SoSeparator* mkroom();

SoSeparator* mkwall()
// create a wall
{
    SoSeparator* wall = new SoSeparator;

    SoCube*  cube = new SoCube;
    cube->height.setValue(0.1);

    SoTranslation*  xt = new SoTranslation;
    xt->translation = SbVec3f( 1.0, 0.0, 1.0 );

    wall->addChild(xt);
    wall->addChild(cube);

    return wall;
```

```
}

SoSeparator* mkroom()
// create a room
{
    SoSeparator*  room = new SoSeparator;
    SoSeparator*  wall = mkwall();

    SoRotation*  xr1 = new SoRotation;
    xr1->rotation.setValue( SbVec3f(1.0,0.0,0.0), -M_PI/2 );
    SoRotation*  xr2 = new SoRotation;
    xr2->rotation.setValue( SbVec3f(0.0,0.0,1.0), -M_PI/2 );

    room->addChild(wall);
    room->addChild(xr1);
    room->addChild(wall);
    room->addChild(xr2);
    room->addChild(wall);

    return room;
}


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();

    // Make a scene
    SoPerspectiveCamera* myCamera = new SoPerspectiveCamera;

    root->addChild(myCamera);

    // Add a directional light
    ///*
    SoDirectionalLight* myDirLight = new SoDirectionalLight;
    myDirLight->direction.setValue( 0, -1, -1 );
    myDirLight->color.setValue( 0, 0, 0.5 );
    root->addChild(myDirLight);
    //*/

    // Add the point light
    /*
    SoPointLight* myPointLight = new SoPointLight;
    myPointLight->location.setValue( 0.8, 0.8, 2 );
    myPointLight->color.setValue( 1, 0, 0 );
    root->addChild(myPointLight);
    */

    // Add the spot light
    /*
    SoSpotLight* mySpotLight = new SoSpotLight;
    mySpotLight->location.setValue( 0.8, 0.8, 2 );
    mySpotLight->direction.setValue( -1, -1, -3 );
    mySpotLight->dropOffRate.setValue(0.5);
    mySpotLight->cutOffAngle.setValue(M_PI/10);
    mySpotLight->color.setValue( 0, 1, 0 );
    root->addChild(mySpotLight);
```

```
    */

    SoMaterial* myMaterial = new SoMaterial;
    myMaterial->diffuseColor.setValue(1.0, 1.0, 1.0);
    root->addChild(myMaterial);

    SoRotation* xr = new SoRotation;
    xr->rotation.setValue( SbVec3f(-1.0,1.0,0.0), -M_PI/6);
    root->addChild(xr);
    root->addChild(mkroom());

    // Create a renderArea in which to see our scene graph.
    // The render area will appear within the main window.
    SoXtRenderArea* myRenderArea = new SoXtRenderArea(myWindow);

    // Make myCamera see everything.
    myCamera->viewAll( root, myRenderArea->getViewportRegion() );

    // Put our scene in myRenderArea, change the title
    myRenderArea->setSceneGraph(root);
    myRenderArea->setTitle("Light");
    myRenderArea->show();

    SoXt::show(myWindow);   // Display main window
    SoXt::mainLoop();

    return 0;
}
```