

Objektorientierte 3D-Programmierung mit Open Inventor

Seminar "Modelling & Rendering"

Abteilung Medieninformatik, Universität Ulm

17. April 2002

Andreas Bentele

andreas.bentele@informatik.uni-ulm.de

Betreuer:

Stefan Fiedler

Zusammenfassung

Zur Programmierung von 3D-Grafik werden heutzutage hauptsächlich professionelle Programmbibliotheken verwendet. Sie ermöglichen die Programmierung von 3D-Modellen und das *Rendern* dieser Modelle. Die bekanntesten Bibliotheken dafür sind *OpenGL* und *DirectX*.

Open Inventor basiert auf *OpenGL*, hat aber ein objektorientiertes Modell, den *Szenengraphen*. Zusätzlich zu den Fähigkeiten von *OpenGL* für das *Rendern* von Szenen bietet *Inventor* eine bessere Unterstützung für Interaktionen mit dem Modell und für das programmatische Verändern der Modelle nach dem *Rendern*. Außerdem gibt es in *Inventor* spezielle Komponenten für die Integration in grafische Benutzungsoberflächen, mit denen der Benutzer komplexe Änderungsoperationen am Modell vornehmen kann. Mit *Inventor* lassen sich somit einfacher interaktive Programme erstellen als mit *OpenGL*, welches sich eher als eine Hardware-unabhängige *Render-Engine* versteht.

1. Einleitung

Zur Programmierung von 3D-Grafikanwendungen gibt es komplexe Programmbibliotheken. Sie ermöglichen die Modellierung von 3D-Grafiken und das Zeichnen dieser Grafiken auf nahezu beliebigen Ausgabegeräten (*Rendern*). Dies zu bewerkstelligen stellt schon eine große Herausforderung dar, da die Modellierung und das *Rendern* auf einer Vielzahl von Grafikkarten mit sehr unterschiedlichen Eigenschaften – von softwareemulierter 3D-Grafik auf Billig-Systemen bis zu hardwarebeschleunigten speziellen 3D-Grafikkarten – eine sehr komplexe Aufgabenstellung ist. Bei solchen Bibliotheken kommt es vor allem auf die Geschwindigkeit beim *Rendern* an, wobei bei der Qualität der Ausgabe durchaus Kompromisse gemacht werden. Aufgrund der Komplexität dieser Aufgabe werden Anwendungsprogramme praktisch immer auf solchen speziellen 3D-Bibliotheken aufsetzen.

Weitergehende wichtige Anforderungen an solche Bibliotheken sind u.a.:

- Unterstützung von Interaktionen durch die Auswertung von Ereignissen des zugrundeliegenden Fenstersystems
- intuitives grafisches Modell
- Möglichkeit für den Austausch von Modelldaten zwischen verschiedenen Programmen und Prozessen
- Unterstützung von Änderungen an den Modelldaten nach dem *Rendern*
- Unterstützung von Animationen

Inventor bietet diese über *OpenGL* hinausgehenden Fähigkeiten, wie in diesem Artikel gezeigt wird.

Der Vortrag und dieser Artikel sollen die Möglichkeiten von *Inventor* zeigen und *Inventor* mit anderen 3D-Bibliotheken vergleichen. Im folgenden wird zunächst gezeigt, wie mit *Inventor* programmiert werden kann. Anschließend werden die wichtigsten Konzepte von *Open Inventor* erklärt. Bei Gelegenheit werde ich die Unterschiede vor allem zu *OpenGL* und *Java 3D* anmerken. In den letzten Abschnitten werde ich auf ein paar fortgeschrittenere Konzepte von *Inventor* eingehen. Abschließend werde ich nochmal die wesentlichen Eigenschaften von *Open Inventor* und die Unterschiede zu den anderen Bibliotheken erläutern.

An dieser Stelle sei angemerkt, dass *Inventor* wie *OpenGL* eine Fülle von Grafikoperationen unterstützt, deren Erklärung den angestrebten Umfang dieses Artikels in jedem Fall sprengen würden. Wer mit *Inventor* programmieren will, dem sei deshalb das Buch „*The Inventor Mentor*“ [1] empfohlen, sowie die Referenz für die *Inventor API* [2]. Fortgeschrittenere Themen im Zusammenhang mit den Erweiterungsmöglichkeiten von *Inventor* werden in [3] behandelt.

Diesen Artikel und meinen Vortrag stütze ich vor allem auf [1].

2. Programmierung mit Inventor

2.1 Das Toolkit

Das Open Inventor Toolkit ist in C++ geschrieben und hat eine C- und eine C++-API. Open Inventor ist Open-Source für Linux und kann daher bei Bedarf an spezielle Anforderungen angepasst werden. Es existieren Versionen für IRIX, Unix, Linux und Windows.

Open Inventor wurde speziell für interaktive Anwendungen entwickelt und enthält daher neben Klassen, die Grafik-Primitiven entsprechen, auch noch Tools, die vor allem die interaktive Arbeit mit 3D-Szenen erleichtern. Auf der Programmiererebene hat Inventor folgende Tools, deren Bedeutungen im Laufe dieses Artikels gezeigt werden:

- *Szenen-Datenbank* zur Speicherung von *Szenengraphen*
- *node kits* zur Programmierung von vordefinierten Knoten-Gruppen
- Manipulatoren (*handle box* und *trackball*) für die Unterstützung von Interaktionen
- verschiedene Komponenten für das *Xt* Fenstersystem (*render area*, *material editor*, *directional light editor* und *examiner viewer*).

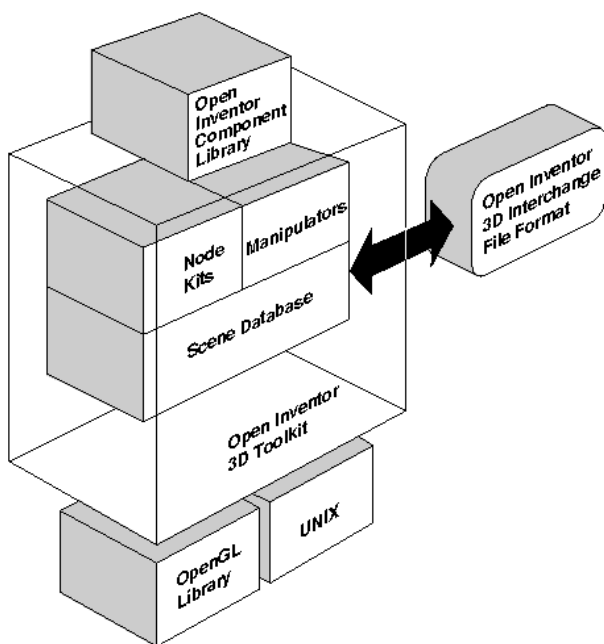


Abbildung 2-1. Architektur von Inventor

Abbildung 2-1 zeigt die Architektur von Inventor. Inventor basiert auf OpenGL und Unix. Der für Anwendungsprogrammierer wohl interessanteste Teil der Architektur ist die *Szenendatenbank*, die den *Szenengraphen* verwaltet. *Node kits* sind ein fortgeschritteneres Konzept für die Strukturierung von Knoten innerhalb der Szenendatenbank,

ein mit der objektorientierten Vererbung vergleichbares Konzept. *Manipulatoren* sind grafische interaktive Komponenten für die interaktive Veränderung der Szene. Sie werden mit der Szene gerendert und können mit einem Zeigegerät verändert werden, z.B. skaliert, gedreht oder verschoben werden. Inventor ist unabhängig von einem bestimmten Fenstersystem, enthält aber in der *Inventor Component Library* Objekte zur Benutzung des Fenstersystems *Xt*, das damit von Inventor derzeit am besten unterstützt wird. Die Component Library kann erweitert werden, sodass auch andere Fenstersysteme benutzt werden können [3]. Zum Austausch mit anderen Programmen und Prozessen existiert ein offenes Dateiformat. Abbildung 2-1 zeigt die Architektur der Bibliothek.

Inventor bildet praktisch fast alle Grafikprimitive von OpenGL irgendwie nach. Die Grafikprimitive ähneln sich sehr stark, sodass die Performance und sonstige Eigenschaften von OpenGL so gut wie möglich ausgenutzt werden können. An dieser Stelle möchte ich einmal kurz und ohne Anspruch auf Vollständigkeit aufzählen, welche Grafikfunktionen Inventor bietet. Diese Aufzählung ist nicht für das Verständnis des Artikels nötig, und eher für etwas fortgeschrittenere Grafikprogrammierer gedacht. Deshalb sind die Funktionen nahezu ohne weitere Erläuterung im folgenden aufgelistet:

- Kamera: zur Einstellung der Betrachtungsrichtung und Projektionsart
- Beleuchtung: Position, Farbe und Art des Lichts
- einfache und komplexe geometrische Formen (Face Set, Indexed Face Set, Triangle Strip Set, Quad Mesh)
- Eigenschaften:
 - Material: Farbe von geometrischen Objekten
 - Zeichenstil: nur Linien vs. gefüllte Flächen
 - Beleuchtungsmodell
 - Umgebungseigenschaften (*Environment*)
 - Shape hints
 - Complexity: zur Verbesserung der Render-Performance
- Material- und Normalen-Bindungen: zur Definition, welches Material bzw. welche Normale eines Arrays zu welcher geometrischen Ecke (*Vertex*) einer geometrischen Form gehört
- Transformationen, z.B. Drehung, Skalierung, Verschiebung

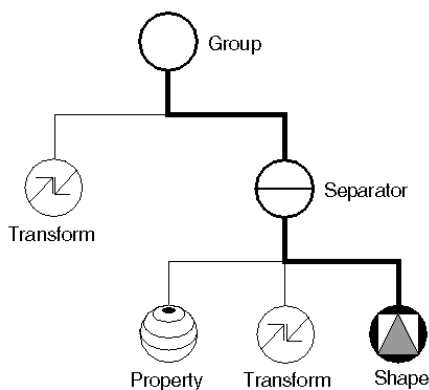
- Text (2D und 3D)
- Texture-Mapping: Bilder (*Textures*) werden über geometrische Formen gelegt, um Objekte realistischer erscheinen zu lassen
- Kurven und Oberflächen (NURBS): durch Anwendung von Interpolation und Bezier-Kurven werden weiche Kurven und Flächen gezeichnet

2.2 Der Szenengraph

Das zentrale Konzept von Inventor ist der *Szenengraph*. Ein Szenengraph ist ein gerichteter azyklischer Graph mit nur einem Wurzelknoten, der alle Objekte, die die Szene in irgendeiner Weise beeinflussen, enthält. Für jede Szene existiert genau ein Szenengraph.

Die *Szenendatenbank* kann einen oder mehrere Szenengraphen verwalten. Die Bezeichnung „Datenbank“ ist eigentlich nicht das richtige Wort, da die minimale Anforderung für Datenbanken, nämlich Daten persistent zu verwalten, bei der Szenendatenbank nicht zutrifft. Auch andere für Datenbanken übliche Konzepte gibt es in Inventor nicht, wie z.B. Transaktionen. Vielmehr ist die Szenendatenbank eine komplexe objektorientierte Datenstruktur im Hauptspeicher.

Die Basis-Datenstruktur der Datenbank ist der *Knoten* (engl. *node*), von dem eine Vielzahl von Klassen abgeleitet sind. Ein Knoten wird als Teil des Szenengraphen von einem oder mehreren Knoten referenziert und er kann selbst einen oder mehrere Knoten referenzieren. Je nach Typ des Knotens enthält ein Knoten Informationen z.B. über die Form eines 3D-Körpers (Kugel, Quader, Kegel, etc.), das Oberflächenmaterial, Transformation, Licht oder Kamera. Auch interaktive Elemente (*Manipulatoren*) und sogar einfache Animationen werden mit Hilfe solcher Knoten abgebildet.



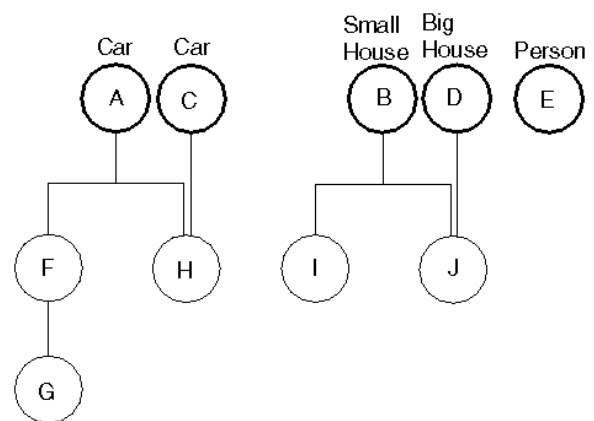
— Path

Abbildung 2-2. Beispiel eines einfachen Szenengraphen

Mehrere Knoten können zu einer *Gruppe* zusammengefasst werden. Eine Gruppe ist ein spezieller Knoten, der Verweise auf einen oder mehrere andere Knoten hat. Abbildung 2-2 zeigt einen einfachen Szenengraphen mit sechs Knoten, zwei davon sind Gruppen.

An die Knoten sind die Typen (Klassen) der Knoten angeschrieben. Die verwendeten Knoten-Symbole sind die in der Inventor-Referenz benutzten. Der Szenengraph wird beim Rendern wie folgt interpretiert: der Szenengraph wird von oben nach unten und von links nach rechts traversiert. Jeder Knoten wird typabhängig gerendert. Das Transformations-Objekt („Transform“) repräsentiert zum Beispiel eine geometrische Transformation und führt beim Rendern zu einer Veränderung der Transformationsmatrix des sogenannten *Render-Status*. Der Render-Status wird beim Rendern der in der Traversierungsreihenfolge folgenden Formen angewendet, sodass immer die zuletzt gesetzte Eigenschaft das Rendern der Form beeinflusst. Das Property-Objekt stellt eine hier nicht näher definierte Eigenschaft wie z.B. eine Material- oder Beleuchtungseigenschaft dar. Beim Rendern werden bei einem Vorkommen eines Property-Objekts die entsprechenden Eigenschaften des Render-Status überschrieben. Das *Shape*-Objekt repräsentiert eine geometrische Form und wird beim Traversieren des Graphen unter Berücksichtigung des aktuellen Render-Status gerendert.

Abbildung 2-3 zeigt eine Szenendatenbank mit mehreren Szenengraphen (Car, Car, Small House, Big House, Person). Zu beachten ist hier, dass z.B. Knoten H von zwei verschiedenen Knoten referenziert wird (und sogar zu zwei verschiedenen Szenengraphen gehört). Dies nennt man *shared instancing*.



...scene database...

Abbildung 2-3 Szenendatenbank mit mehreren Szenengraphen

Das shared instancing ist intuitiv: um ein 3D-Objekt an verschiedenen Stellen und mit ggf. unterschiedlichen Eigenschaften zu rendern, ist es nicht einsichtig, warum

man die Knoten dafür zweimal erzeugen soll. Stattdessen referenziert man die Objektgruppe des 3D-Objekts dafür mehrmals. Außerdem ist diese Methode ökonomisch im Speicherverbrauch. Trotz der Möglichkeit für shared instancing dürfen keine Zyklen im Szenengraphen auftreten. Ein Knoten kann zwar mehrere Eltern haben, darf aber nicht Kind von sich selbst oder von seinen Kindern sein.

2.3 Ein Beispiel

Ein kurzes aber komplettes Programmbeispiel soll die Programmierung mit Inventor zeigen.

Die einzelnen Schritte bei der Programmierung einer Szene sind normalerweise die folgenden:

- Erzeugen eines Fensters und eines Renderbereichs, in dem die Szene gerendert werden soll (hier ein Objekt vom Typ *SoXtRenderArea*)
- Konstruktion des Szenengraphen
- Rendern des Szenengraphen innerhalb des Renderbereichs des Fensters

Folgendes Beispielprogramm (in C++) rendert einen Konus:

```
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/Xt/SoXt.h>

main(int , char **argv)
{
    // Initialisiere Inventor. Gibt ein Fenster
    // zurück.
    widget mywindow = SoXt::init(argv[0]);
    if (mywindow == NULL) exit(1);

    // Erzeuge eine Szene mit rotem Konus
    SoSeparator *root = new SoSeparator();
    SoPerspectiveCamera *myCamera
        = new SoPerspectiveCamera();
    SoMaterial *myMaterial = new SoMaterial();
    //für die Speicherverwaltung von Inventor
    //nötig: Referenzierung des wurzelknotens
    root->ref();
    root->addChild(myCamera);
    root->addChild(new SoDirectionalLight);
    myMaterial->
        diffuseColor.setValue(1.0,0.0,0.0);
    root->addChild(myMaterial);
    root->addChild(new SoCone);

    SoXtRenderArea *myRenderArea
        = new SoXtRenderArea(mywindow);

    // Kameraposition setzen
    myCamera->viewAll(root,
        myRenderArea->getViewportRegion());

    // Szenengraph mit dem Renderbereich
    // verknüpfen
    myRenderArea->setSceneGraph(root);
    // Fenstertitel setzen
    myRenderArea->setTitle("Hello Cone");
    myRenderArea->show();
}
```

```
SoXt::show(mywindow); // Fenster anzeigen
SoXt::mainLoop(); //Inventor-Ereignisschleife
}
```

Zunächst wird Xt initialisiert, womit implizit auch die Szenendatenbank initialisiert wird. Anschließend wird der Wurzelknoten erzeugt (vom Typ *SoSeparator*, abgeleitet von *SoGroup*). Dieser Gruppe werden mit der Methode *addChild* ein Kamera-, Licht-, Material- und der Konus-Objekt hinzugefügt. Das Programm erzeugt also folgenden Szenengraphen:

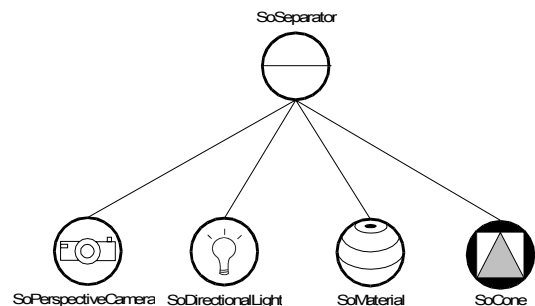


Abbildung 2-4. Szenengraph: Konus

Anschließend wird die Szene in einer *SoXtRenderArea* gerendert, wie die Bildschirmkopie in Abbildung 2-5 zeigt.

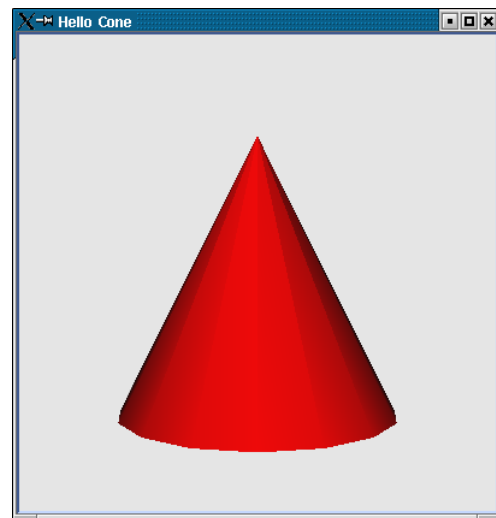


Abbildung 2-5. Bildschirmkopie des Beispiels

Bei der Konstruktion einer Szene sind Kamera und Licht von großer Bedeutung. Die Kameraposition bestimmt, von welcher Position aus und mit welcher Perspektive man die Szene betrachtet. In diesem Beispiel wird die Kamera so positioniert, dass die ganze Szene sichtbar ist. Ohne Beleuchtung ist eine Szene praktisch unsichtbar, deshalb sollte in jedem Szenengraphen mindestens ein Beleuchtungsobjekt existieren. Das Material-Objekt wird in dieser Szene dazu benutzt, den Konus rot einzufärben.

2.4 Engines für einfache Animationen

Folgendes Beispiel soll zeigen, wie eine in Inventor als *Engine* bezeichnete Animationskomponente für eine kontinuierliche Rotation des Konus verwendet werden können. Dazu wird in den Szenengraph ein **SoRotationXYZ**-Objekt eingefügt und dessen Attribut **angle**, das den Winkel einer Rotation enthält, mit einem Engine-Objekt (hier **SoElapsedTime**) verknüpft.

SoElapsedTime reagiert auf Veränderungen der Systemzeit und ändert als Reaktion darauf das verknüpfte Attribut **angle**. Der Winkel wird dabei ständig erhöht, sodaß die Veränderung eine kontinuierlichen Drehung des Konus bewirkt. Bei jeder Veränderung dieses Attributs wird die Szene automatisch neu gerendert (i.A. wird nur der Teil neu gerendert, der sich durch die Veränderung des Szenengraphen tatsächlich verändert hat).

Einfache Animationen können in Inventor also mit Standard-Objekten erzeugt werden. Das hat unter anderem auch den Vorteil, dass die Animation als Teil des Szenengraphen in Dateien geschrieben werden kann. Außerdem ist für einfache Animationen relativ wenig Code notwendig.

Das Beispiel ist im folgenden abgedruckt, ebenso der resultierende Szenengraph (*include*-Anweisungen werden in den folgenden Code-Sequenzen des Artikels der Einfachheit halber weggelassen).

```
main(int , char **argv)
{
    widget mywindow = SoXt::init(argv[0]);
    if (mywindow == NULL) exit(1);

    SoSeparator *root = new SoSeparator;
    root->ref();
    SoPerspectiveCamera *myCamera
    = new SoPerspectiveCamera;
    root->addChild(myCamera);
    root->addChild(new SoDirectionalLight);

    SoRotationXYZ *myRotXYZ = new SoRotationXYZ;
    root->addChild(myRotXYZ);
    // Rotation um X-Achse definieren
    myRotXYZ->axis = SoRotationXYZ::X;
    SoElapsedTime *myCounter = new SoElapsedTime;
    // Engine mit Attribut "angle" verbinden
    myRotXYZ->angle.connectFrom(
        &myCounter->timeOut);

    SoMaterial *myMaterial = new SoMaterial;
    myMaterial->
        diffuseColor.setValue(1.0,0.0,0.0);
    root->addChild(myMaterial);
    root->addChild(new SoCone);

    SoXtRenderArea *myRenderArea
    = new SoXtRenderArea(mywindow);
    myCamera->viewAll(root,
        myRenderArea->getViewPortRegion());
    myRenderArea->setSceneGraph(root);
    myRenderArea->setTitle("Engine Spin");
    myRenderArea->show();

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

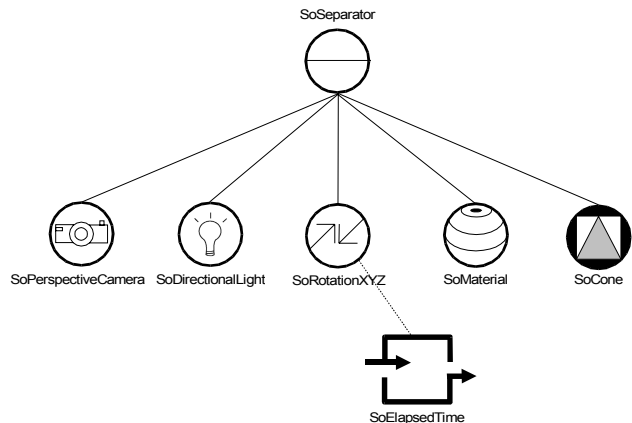


Abbildung 2-6. Szenengraph: Konus mit Rotation

Man beachte den Unterschied zwischen Knoten und Engines. Engines sind ein Beispiel für Objekte der Szenendatenbank, die keine Knoten sind. Dies bedeutet insbesondere, dass Engines nicht Teil des Szenengraphen sind. Für die Praxis bedeutet das, dass man beim Überprüfen, ob der Szenengraph azyklisch ist, auf die Betrachtung der Engines verzichten kann. Außerdem spielen Engines auch keine Rolle bei der Traversierung des Szenengraphen, was zum Beispiel beim Rendern geschieht.

2.5 Actions

Bisher wurde gezeigt, wie ein Szenengraph konstruiert ist und welche Bestandteile er hat. Wie aber wird in Szenengraph verarbeitet, wenn mehr als ein Knoten dabei betrachtet werden muss? Das trifft zum Beispiel für das Rendern des Szenengraphen zu, wo jeder einzelne Knoten des Szenengraphen traversiert und gerendert werden muss. Dazu gibt es in Inventor *Actions*.

Eine Action ist ein Objekt, das einen komplexen Algorithmus auf der Basis eines Teilgraphen definiert. Eine Action wird auf einen Knoten oder einen *Pfad* angewendet. Ein Pfad ist ein Weg von der Wurzel eines Szenengraphen zu einem bestimmten Knoten.

Bei Anwendung einer Action auf einen Knoten wird der Teilgraph, dessen Wurzel dieser Knoten ist, traversiert. Die Reihenfolge der Traversierung ist vom Typ der Action abhängig, meist wird aber von oben nach unten und von links nach rechts traversiert. Dabei verwaltet die Datenbank einen Traversierungsstatus, der während der Traversierung durch die Knoten verändert werden kann, abhängig von deren speziellen Verhaltensweise für diese Action. Der Objekttyp des Traversierungsstatus wird vom Typ der Action bestimmt.

Praktisch alle Operationen, die eine Traversierung des Szenengraphen erforderlich machen, sind in Inventor Actions. Folgende Actions können auf Knoten angewendet werden:

- Rendern des Subgraphen
- Berechnung einer *bounding box* (eine bounding box wird für Optimierungen z.B. beim Rendern benötigt; die bounding box ist eine quaderförmige geometrische Struktur, die angibt, innerhalb welcher Koordinaten eine Gruppe von geometrischen Objekten gerendert wird)
- Auswahl einzelner Objekte und ganzer Objektgruppen
- Änderung von Objekten
- Suchen nach einzelnen Objekten
- Drucken
- Lesen von und Schreiben in eine Datei
- Action für die Ausführung von Callbacks einzelner Knoten (der Einsatz von Callback-Knoten wird im Zusammenhang mit der Verwendung von OpenGL in Inventor in Abschnitt 3.4 anhand eines Beispiels gezeigt)

Als Beispiel soll hier die Action, die für das Rendern einer Szene zuständig ist, näher beschrieben werden. Hier enthält der Traversierungsstatus die aktuellen Parameter für das Rendern von Objekten. Dazu gehören z.B. die aktuelle Transformations-Matrix, Materialeigenschaften, das Beleuchtungsmodell, der Zeichnungsstil und der Text-Font. Jedesmal wenn z.B. ein *SoMaterial*-Knoten traversiert wird, so werden die Materialeigenschaften des Traversierungsstatus überschrieben. *SoTransformation*-Knoten konkatenieren die dadurch gegebene Transformation mit der Transformationsmatrix des Traversierungsstatus. Shape-Knoten ändern den Traversierungsstatus nicht, sondern führen zu einem Rendern der geometrischen Form. Beim Rendern eines Shape-Objekts wird jeweils der aktuelle Traversierungsstatus benutzt.

2.6 Separatoren

Um Seiteneffekte bei der Anwendung von Actions außerhalb einer Knotengruppe zu vermeiden, werden spezielle Gruppenknoten verwendet, die *Separatoren*.

Bei Anwendung einer Action auf eine Gruppe werden normalerweise einfach alle durch die Gruppe referenzierten Knoten in einer bestimmten Reihenfolge traversiert. Ein Separator hat die selbe Funktion, darüber hinaus jedoch sichert er den Traversierungsstatus vor der Traversierung der referenzierten Knoten und stellt ihn danach wieder her. Somit beeinflussen die Kindknoten des Separators den Traversierungsstatus nur unterhalb des Separators, nicht jedoch oberhalb oder rechts des Separators.

Speziell wird ein Separator meist auch als Wurzel eines Szenengraphen verwendet, damit ein Rendervorgang nicht vom vorhergehenden Rendervorgang beeinflusst wird.

Daneben gibt es noch einige andere von *SoGroup* abgeleitete Gruppen-Knoten:

- *SoSwitch*: ein Index-Attribut bestimmt, welcher der Kindknoten beim Rendern berücksichtigt wird.
- *SoBlinker*: abgeleitet von *SoSwitch*; das Index-Attribut wird zeitgesteuert verändert; damit sind einfache Animationen realisierbar
- *SoLevelOfDetail*: ermöglicht das Rendern ein und desselben Objekts in verschiedenen Detailstufen, abhängig von der Größe des Zeichnungsbereichs. Dies wird realisiert, indem diese Gruppe für jede Detailstufe jeweils einen Subgraphen enthält. Beim Rendern wird nur einer der Subgraphen gerendert. Damit lässt sich die für das Rendern benötigte Zeit auf Kosten der Detailgenauigkeit optimieren.
- *SoSelection*: Objektgruppe, die sich mit einem Zeigegerät selektieren lässt

2.7 Pfade

Um ein bestimmtes Objekt einer 3D-Grafik zu isolieren, verwendet Inventor *Pfade*. Da der Weg von der Wurzel zu einem bestimmten Knoten des Szenengraphen nicht eindeutig ist, wie das in einem Baum der Fall wäre, muss dieser Pfad durch eine Aufzählung der Knoten entlang des Pfades spezifiziert werden.

Pfade werden durch Auswahl- oder Such-Actions berechnet und zurückgegeben. Klickt der Benutzer ein Objekt am Bildschirm mit der Maus an, so wird das Objekt ausgewählt. Der *Selection* Knoten (Selection: siehe voriger Abschnitt) verwaltet eine Liste von Pfaden der gegenwärtig ausgewählten Objekte.

Wie schon zuvor erwähnt, können alle oben beschriebenen Actions, die auf einzelnen Knoten angewendet werden können, auch auf Pfade angewendet werden.

2.8 Speicherverwaltung

Obwohl Inventor in C++ geschrieben wurde, ist in Inventor eine komfortable Speicherverwaltung integriert, die ähnlich wie bei Java ein explizites Freigeben von Objekten unnötig macht. Der Programmierer sollte deshalb auf keinen Fall selbst Objekte mit *delete* aus dem Speicher löschen. Der für die Speicherverwaltung verwendete Mechanismus beruht wie bei Java auf einer Verwaltung von Referenzzählern für Objekte.

Die Art, wie Inventor den Speicher verwaltet, ist für die Anwendungs-Programmierung essenziell. Der Programmierer muss die Speicherverwaltungsmechanismen genau

kennen und berücksichtigen. Für eine ausführliche Beschreibung dieser Mechanismen verweise ich auf [1].

Für die Beispiele dieses Artikels genügt es zu wissen, dass ein Wurzelknoten eines Szenengraphen durch keinen anderen Knoten referenziert wird und deshalb explizit im Anwendungsprogramm referenziert werden muss. Ein nicht referenzierter Knoten kann nicht gerendert werden. Folgende Anweisung inkrementiert den Referenzzähler für den Wurzelknoten:

```
root->ref();
```

2.9 Node Kits

Mit *Node kits* kann man eine Menge von Knoten zu einem Knoten zusammenfassen. Von aussen betrachtet ist ein node kit selbst ein Knoten, der wie andere Knoten in den Szenengraph eingefügt werden kann. Ein node kit kann als eine black box gesehen werden, die einen ganzen Subgraphen hinter einem Knoten verbirgt.

Der Zweck von node kits ist, einer Gruppe von Knoten zusätzliche Methoden zu geben, und, wie bei der objektorientierten Vererbung, Information zu kapseln und zu verstecken.

Einen ähnlichen Effekt wie durch die Anwendung von node kits kann man erreichen, indem man eine neue Node-Klasse durch Vererbung von einer anderen Node-Klasse erzeugt. Die Subklasse definiert intern die gewünschte Knotenstruktur, und von aussen betrachtet ist das Objekt ein normaler Knoten.

Beide Konzepte, node kits und in oben beschriebener Weise die objektorientierte Vererbung, gleichen sich einander. Der Nachteil bei der Methode mit Vererbung ist, dass die Traversierung der Knoten innerhalb des Knotens nachgebildet werden muss, da Inventor die Strukturen innerhalb eines Knotens nicht kennt. Node kits jedoch sind so konstruiert, dass sie die Knoten innerhalb traversieren können. Es ist also einfach weniger Aufwand, node kits zu verwenden als Vererbung.

Ein node kit enthält normalerweise bei der Instanziierung schon einige Knoten, und es können bei Bedarf weitere Knoten hinzugefügt werden. Die Art der Knoten, die hinzugefügt werden können, wird durch das node kit definiert. Die Anordnung der Knoten wird vom node kit organisiert. Ein node kit kann auch weitere node kits enthalten, sodaß hierarchische Strukturen mit node kits realisierbar sind.

Node kits bieten Methoden zur Vereinfachung der Erzeugung von Knoten des node kits. Damit wird der Code kürzer und besser lesbar. Eigene node kit Klassen können durch Vererbung erstellt werden [3].

Obwohl auf node kits verzichtet werden könnte, sind sie in der Praxis ein wichtiges Hilfsmittel zur Strukturierung von

Szenengraphen, und es wird stark empfohlen, diese zu verwenden.

2.10 Manipulatoren

Inventor wurde speziell im Hinblick auf die Unterstützung von Interaktivität entworfen. Ein wichtiges Element dabei sind *Manipulatoren*.

Ein Manipulator ist ein spezieller Knoten, der auf Benutzerschnittstellen-Ereignisse reagiert und direkt vom Benutzer editiert werden kann. Typischerweise werden Manipulatoren in der Szene gerendert, z.B. als greifbare Box.

Der Benutzer kann durch entsprechende Interaktionen mit dem Manipulator ein Objekt drehen, verschieben und skalieren. Änderungen an solchen Manipulatoren werden sofort umgesetzt in entsprechende Änderungen im Szenengraphen. In der Folge werden betroffene Teile des Szenengraphen invalidiert und neu gerendert, sodaß die Wirkung der Interaktion sofort am Bildschirm nachvollzogen werden kann.

2.11 Komponenten-Bibliothek

Um das Rendern der Szenen bewerkstelligen zu können, ist eine Integration von Inventor in ein Fenstersystem (z.B. X) nötig. Diese Integration wird durch die *Inventor Component Library* realisiert. Die Bibliothek hat folgende Teile:

- Fensterobjekt für das Rendern von Szenen
- Main loop und Initialisierungsroutinen
- Ereignis-Verarbeitung
- Komponenten: Editoren (*material editor*, *directional light editor*) und Betrachtungskomponenten (*fly viewer*, *examiner viewer*)

Die Bibliothek kann erweitert werden, um z.B. weitere Fenstersysteme zu unterstützen oder um weitere Eigenschaftseditoren hinzuzufügen [3]. Die Eigenschaftseditoren und die erweiterten Betrachtungskomponenten stellen im Vergleich zu OpenGL eine Zusatzfunktion dar und erleichtern den Einstieg in die Programmierung mit Inventor sowie die schnelle Erstellung von grafischen Benutzungsoberflächen für Inventor-Programme.

Ich bezweifle jedoch die Eignung dieser Komponenten für professionelle Anwendungsprogramme. Für Prototypen und einfachere Demonstrationen sind sie allerdings durchaus geeignet.

3. Fortgeschrittene Programmierung mit Inventor

3.1 Ereignisbearbeitung

Ähnlich wie bei einem Fenstersystem müssen auch bei Inventor Ereignisse des Fenstersystems verarbeitet werden. Dies sind Tastaturereignisse und Ereignisse von Zeigegeräten (z.B. Maus). Die Verarbeitung von Ereignissen von Zeigegeräten ist in 3D-Grafiksystemen sehr komplex. Vor allem die Zuordnung eines Pixels auf dem Bildschirm zu einem geometrischen Objekt ist eine schwierige Aufgabe, was für die Selektion von Objekten mit Zeigegeräten benötigt wird.

OpenGL bietet keine Funktionen für die Ereignisbearbeitung, und in OpenGL ist es relativ kompliziert, ein Pixel des gerenderten Bildes einem grafischen Objekt zuzuordnen.

In diesem Abschnitt soll gezeigt werden, wie Inventor Ereignisse des X Window Systems übersetzt und für ein bestimmtes Ereignis den entsprechenden Event-Handler für die Ereignisbearbeitung findet.

Abbildung 2-7 zeigt, wie Ereignisse von Inventor übersetzt werden. X Ereignisse werden zunächst an die *RenderArea* (Fensterobjekt, innerhalb dem das Bild gerendert wird) des Fenstersystems weitergereicht. Dort werden sie von dem für das Fenstersystem spezifischen *Event Translator* in Inventor-Ereignisse übersetzt und an den *Scene Manager* von Inventor weitergereicht.

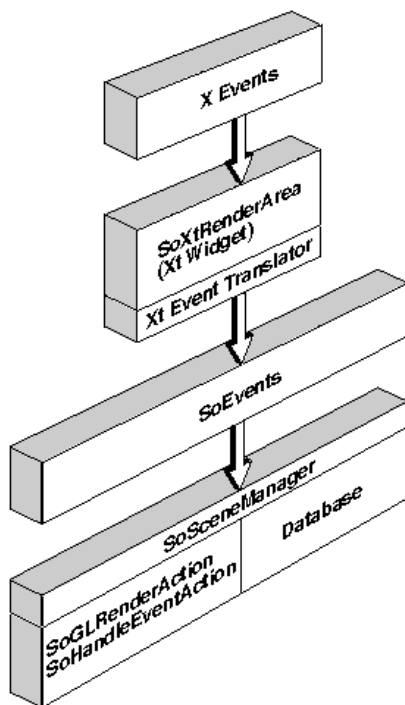


Abbildung 3-1. Ereignisbearbeitung in Inventor

Ein Inventor-Ereignis enthält mindestens folgende Felder:

- Ereignis-Typ
- Zeit des Auftretens
- Cursor Position beim Auftreten
- Status der Modifier-Tasten (Shift, Control, Alt)

Für jede Zeigegeräte-Kategorie gibt es einen Event-Typ. Je nach Ereignis-Typ kommen weitere Felder hinzu. Inventor kann um weitere Ereignis-Typen erweitert werden, sodass Inventor auch für neuartige Zeigegeräte angepasst werden kann [3].

Der *Scene Manager* ist ein Objekt, das die Fenstersystem-abhängigen Zeichenbereiche mit dem ansonsten Fenstersystem-unabhängigen Inventor verbindet. Der Scene Manager verwaltet den Szenengraph und besorgt das Rendern und die Ereignisbearbeitung. Der Scene Manager selbst ist unabhängig vom Fenstersystem.

Ereignisse werden in Inventor an die 3D-Objekte der Szenendatenbank verteilt. Manipulatoren verarbeiten Ereignisse, wohingegen Form-, Eigenschafts-, Transformations-, Licht- und Kamera-Objekte Ereignisse ignorieren.

Es gibt vier Möglichkeiten für die Ereignisbehandlung in Inventor-Applikationen:

1. Automatische Ereignisbehandlung des Scene Managers, bei der bestimmte Arten von Knoten Ereignisse behandeln.
2. Event callback Knoten: ein spezieller Knoten im Szenengraph, der Ereignisse über eine Callback Routine behandelt
3. Überschreiben des Ereignisbehandlungs-Mechanismus von Inventor. Alle X Events werden direkt zur Applikation weitergegeben; nicht empfohlen, da Fenstersystem-abhängig.
4. Generic callback Mechanismus

Bei der automatischen Ereignisbehandlung wird eine spezielle Action (*SoHandleEventAction*) auf den Wurzelknoten des Szenengraphen angewendet. Der Szenengraph wird traversiert, und nach Knoten gesucht, die das Ereignis bearbeiten können (z.B. ein Manipulator). Wird ein solcher gefunden, so stoppt die Traversierung und das Ereignis wird dem Knoten zur Bearbeitung übergeben.

Ein Knoten kann das Ereignisbehandlungssystem anweisen, alle darauffolgende Ereignisse bis auf Widerruf an ihn zu schicken, und nicht an andere Knoten. Dies wird *grabbing* genannt. Dazu wird die Methode *setGrabber* der Handle Event Action aufgerufen. Zum Aufheben dieses Zustands wird die Methode *releaseGrabber* aufgerufen.

Grabbing wird z.B. von Manipulatoren verwendet, wenn alle Maus-Ereignisse nur für den Manipulator bestimmt sein sollen, nachdem der Manipulator angeklickt wurde, aber der Mausknopf noch nicht losgelassen wurde.

3.2 Dateiformat

Inventor bietet ein Dateiformat, das es ermöglicht, Szenengraphen zu laden und zu speichern. Inventor kann Pfade, durch einen Knoten definierte Subgraphen und komplette Szenengraphen laden und speichern. Mit Dateien ist es möglich, Szenengraphen zwischen Prozessen und Programmen auszutauschen. Ebenso sind Kopier/Einfüge-Operationen über eine Zwischenablage realisierbar. Durch Programme erstellte Szenengraphen lassen sich dauerhaft speichern. Eine weitere öfters verwendete Anwendung von Dateien ist die Erstellung von grafischen Modellen direkt in Dateiform. Dies kann sowohl von Hand mit einem Texteditor geschehen als auch durch Konvertierung von anderen Grafik-Dateiformaten.

OpenGL kann Modelle nicht in Dateien speichern, und das wäre auch nicht zu erwarten, da OpenGL keine 3D-Objekte kennt.

In Inventor gibt es ein binäres Dateiformat und ein lesbares ASCII-Dateiformat. Das ASCII-Dateiformat ist bei Kenntnis des Aufbaus von Szenengraphen schnell zu durchschauen. Beim Betrachten einer Datei erkennt man relativ leicht die Struktur des Szenengraphen und die Werte der Eigenschaften der einzelnen Knoten werden im Klartext dargestellt.

Aus heutiger Sicht erscheint mir das Format nicht mehr zeitgemäß, XML wäre da sicher angebrachter, vor allem weil es dafür validierende Parser, spezialisierte Editoren und andere nützliche Tools gibt. Denkbar wäre zum Beispiel auch, einfachere Operationen am Szenengraphen als XSL-Transformationen zu definieren (*XSL* ist eine auf XML basierende Sprache, mit der man Transformationen von XML-Dokumenten definieren kann). Ein Praktikum der TU Wien hatte zur Aufgabe, eine Inventor-Action zu definieren, die einen Szenengraphen als XML-Datei speichert. Weitergehende Anstrengungen in diese Richtung habe ich nicht recherchieren können.

Eine Untermenge des Inventor-Dateiformats ist übrigens die Basis für die *Virtual Reality Modeling Language* (VRML), ein 3D-Grafikdateiformat für das Internet und Intranet. Mit *X3D* wird zur Zeit ein neues 3D-Grafikformat definiert, das die Funktionalität von VRML haben soll, aber ein XML-Format ist.

Hier nun ein kurzes Beispiel für eine Inventor-Datei im Ascii-Format:

```
Separator {
  PerspectiveCamera {
    position 0 0 9.53374
    aspectRatio 1.09446
```

```
nearDistance 0.0953375
farDistance 19.0675
focalDistance 9.53374
}
directionalLight {
}
Transform {
  rotation -0.189479 0.981839 -0.00950093
0.102051
  center 0 0 0
}
DrawStyle {
}
Separator {
  LightModel {
    model BASE_COLOR
  }
  Separator {
    Transform {
      translation -2.2 0 0
    }
    BaseColor {
      rgb .2 .6 .3 # chartreuse
    }
    Sphere { }
  }
  Separator {
    BaseColor {
      rgb .6 .3 .2 # rust
    }
    Sphere { }
  }
  Separator {
    Transform {
      translation 2.2 0 0
    }
    BaseColor {
      rgb .3 .2 .6 # violet
    }
    Sphere { }
  }
}
}
```

Der Leser betrachte es als kleine Übung, den Szenengraphen, der durch die Datei dargestellt wird, zu zeichnen. Da die Syntax relativ einfach aus dem Beispiel zu erkennen ist, verzichte ich auf eine genauere Beschreibung. Eine genauere Beschreibung des Dateiformats und der Syntax bietet [1], die Referenz ([2]) enthält für jede Klasse die Entsprechungen und Default-Werte im Dateiformat.

3.3 Benutzung von Inventor mit OpenGL

Eine interessante Möglichkeit, nämlich die Benutzung von OpenGL-Befehlen in Inventor-Programmen, möchte ich zum Schluss dieses Artikels behandeln.

Aus ähnlichen Gründen wie man Hochsprachen-Programme mit Assembler-Anteilen schreibt, verwendet man OpenGL-Anteile in Inventor-Programmen. Für OpenGL spricht vor allem die höhere Flexibilität sowie die direkte Zugriffsmöglichkeit auf den frame buffer. Für sich schnell ändernde Szenen ist OpenGL ebenfalls die bessere Wahl. Manchmal will man aber einfach existierenden OpenGL-Code in Inventor-Programme integrieren, damit man den Code nicht extra für Inventor portieren muss.

Es gibt drei verschiedene Möglichkeiten, OpenGL in Inventor-Programmen zu verwenden:

- OpenGL-Code in den Render-Methoden von Subklassen von Inventor-Klassen [3]. Dies ist die bevorzugte Methode.
- OpenGL-Code in Callback-Routinen: in den Szenengraphen wird ein Callback-Knoten eingefügt. Beim Rendern des Szenengraphen wird die Callback-Routine aufgerufen.
- gemischte Anwendung von OpenGL und Inventor beim Rendern in ein GLX-Fenster.

Im Folgenden soll die Methode mit Callback-Routine näher betrachtet werden.

Für das Verständnis der folgenden Abschnitte sind ein paar Kenntnisse über OpenGL notwendig. OpenGL funktioniert ähnlich einem endlichen Automat. Dazu verändern OpenGL-Befehle intern Status-Variablen. Soweit von der Grafik-Hardware unterstützt, werden die Status-Variablen der Hardware entsprechend gesetzt. Beispiele für OpenGL-Status-Variablen sind die aktuelle Transformationsmatrix, die Linienattribute und der Index der aktuellen *display list* (OpenGL kann eine Folge von OpenGL-Befehlen in einer *display list* abspeichern und später wieder abspielen; da mehrere *display lists* existieren, gibt es einen Index, der die aktuell verwendete *display list* definiert).

Da Inventor auf OpenGL basiert, verändert Inventor indirekt OpenGL-Status-Variablen. Wenn Inventor und OpenGL gemeinsam in Anwendungsprogrammen benutzt werden, so erzeugt die direkte Verwendung von OpenGL-Befehlen Seiteneffekte durch die Veränderung von Status-Variablen. Da Inventor OpenGL nur für das Rendern benutzt, betreffen die Seiteneffekte ebenfalls nur das Rendern, nicht jedoch z.B. Interaktionen und Animationen. Es existiert eine genaue Liste über die Status-Variablen, die von Inventor verändert werden ([1], S. 454 ff).

Um Seiteneffekte zu vermeiden, sollte sehr behutsam mit Veränderungen von Status-Variablen vorgegangen werden. Bei Callback-Routinen sollten die Status-Variablen jeweils zu Beginn gesichert und am Ende der Routine wieder ihr ursprünglicher Zustand hergestellt werden. Dafür existieren die beiden OpenGL-Befehle `pushAttributes()` und `popAttributes()`, mit denen die komplette Menge von Status-Variablen auf einen Stack gelegt bzw. wieder geholt werden können. Damit wird die Sichtbarkeit der Veränderungen von Status-Variablen innerhalb der Callback-Routine auf die Callback-Routine eingeschränkt und damit Seiteneffekte auf Inventor verhindert.

Für die Methode mit Callback-Routine gibt es den speziellen Knotentyp *SoCallback*. Wird eine Action auf den Szenengraphen angewendet, so wird bei dem Auftreten

eines *SoCallback*-Knotens dessen registrierte Callback-Routine ausgeführt. Da jede Action zu einem Aufruf der Callback-Routine führt und in diesem Fall nur die *SoGLRenderAction* bearbeitet werden soll, muß innerhalb der Callback-Routine noch der Typ der Action abgefragt werden:

```
if (action->isOfType(
    SoGLRenderAction::getClassTypeId())) {
    ... execute rendering code ...
}
```

Je nachdem, ob die Callback-Routine eine relativ statische oder eine sich stark verändernde Szene zeichnet, macht es Sinn, die OpenGL-Befehle zu cachen. Inventor kann daran gehindert werden, einen Cache aufzubauen, indem man zu Beginn der Callback-Routine den Cache invalidiert mit:

```
void myCallback(void *myData, SoAction *action)
{
    if (action->isOfType(
        SoGLRenderAction::getClassTypeId())) {
        SoCacheElement::invalidate(
            action->getState());
        ... execute rendering code ...
    }
}
```

Zu beachten ist außerdem bei der Verwendung von OpenGL *display lists*, dass Inventor ebenfalls schon eine *display list* geöffnet haben könnte, da Inventor *display lists* für das Caching verwendet. In OpenGL darf aber nicht mehr als eine *display list* gleichzeitig geöffnet sein, sodass im allgemeinen in Callback-Routinen auf *display lists* verzichtet werden muss.

Ein Beispielprogramm soll die gemeinsame Verwendung von OpenGL und Inventor zeigen. Das Programm rendert eine Szene mit zwei Körpern und einem Fußboden. Die Körper werden sind Inventor modelliert, der Fußboden wird direkt mit OpenGL gerendert. Abbildung 3-2 zeigt die Ausgabe des Programms.

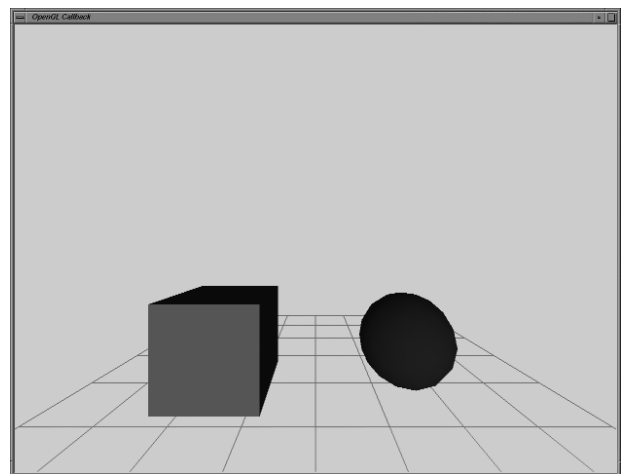


Abbildung 3-2. Szene, gerendert mit Inventor und OpenGL

Im folgenden ist der Sourcecode des Programms komplett abgedruckt. Wichtig ist, die Struktur des Programms zu verstehen. Wer OpenGL nicht kennt, kann die einzelnen GL-Befehle getrost überlesen, es dürfte jedoch klar sein, wie das Programm arbeitet.

```
// geometrische Daten für Fußboden
float floorObj[81][3];

// erzeuge Szene mit zwei Objekten und Licht
void buildScene(SoGroup *root) {

    // Beleuchtung
    root->addChild(new SoLightModel);
    root->addChild(new SoDirectionalLight);

    // Transformation für würfel definieren
    SoTransform *myTrans = new SoTransform;
    myTrans->translation.setValue(-2.0, -2.0,
0.0);
    root->addChild(myTrans);

    // Farbe des würfels
    SoMaterial *myMtl = new SoMaterial;
    myMtl->diffuseColor.setValue(1.0, 0.0, 0.0);
    root->addChild(myMtl);

    // würfel
    root->addChild(new SoCube);

    // Transformation für Kugel definieren
    myTrans = new SoTransform;
    myTrans->translation.setValue(4.0, 0.0, 0.0);
    root->addChild(myTrans);

    // Farbe der Kugel
    myMtl = new SoMaterial;
    myMtl->diffuseColor.setValue(0.0, 0.0, 1.0);
    root->addChild(myMtl);

    // Kugel
    root->addChild(new SoSphere);
}

// geometrische Daten für Fussboden berechnen
// und in Array speichern
void buildFloor() {
    int a = 0;

    for (float i = -5.0; i <= 5.0; i += 1.25) {
        for (float j=-5.0; j<=5.0; j+=1.25, a++) {
            floorObj[a][0] = j;
            floorObj[a][1] = 0.0;
            floorObj[a][2] = i;
        }
    }
}

// Rendern des Fussbodens mit OpenGL
void drawFloor() {
    int i;

    // beginne Linienzug
    glBegin(GL_LINES);

    // Koordinaten (Vertex) des Linienzugs
    // definieren
    for (i=0; i<4; i++) {
        glVertex3fv(floorObj[i*18]);
        glVertex3fv(floorObj[(i*18)+8]);
        glVertex3fv(floorObj[(i*18)+17]);
        glVertex3fv(floorObj[(i*18)+9]);
    }
    glVertex3fv(floorObj[i*18]);
    glVertex3fv(floorObj[(i*18)+8]);

    // beende Linienzug
    glEnd();
}
```

```
// analog wie oben
glBegin(GL_LINES);
for (i=0; i<4; i++) {
    glVertex3fv(floorObj[i*2]);
    glVertex3fv(floorObj[(i*2)+72]);
    glVertex3fv(floorObj[(i*2)+73]);
    glVertex3fv(floorObj[(i*2)+1]);
}
glVertex3fv(floorObj[i*2]);
glVertex3fv(floorObj[(i*2)+72]);
glEnd();
}

// Callback-Routine für das Rendern des
Fußbodens
void myCallbackRoutine(void *, SoAction *) {
    // aktuelle Transformationsmatrix sichern
    glPushMatrix();
    // neue Transformationsmatrix festlegen
    glTranslatef(0.0, -3.0, 0.0);
    // Farbe in RGB definieren
    glColor3f(0.0, 0.7, 0.0);
    // Linienbreite
    glLineWidth(2);
    // Beleuchtung abschalten
    glDisable(GL_LIGHTING);
    // Fussboden zeichnen
    drawFloor();
    // Beleuchtung einschalten
    glEnable(GL_LIGHTING);
    glLineWidth(1);
    // Transformationsmatrix wieder herstellen
    glPopMatrix();
}

main(int, char **)
{
    // Initialisiere Inventor
    Widget myWindow = SoXt::init("Inventor+GL
Beispiel");

    buildFloor();

    // Erzeuge einfachen Szenengraphen mit Kamera
    // und einem SoCallback-Knoten

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

    SoPerspectiveCamera *myCamera
    = new SoPerspectiveCamera;
    myCamera->position.setValue(0.0, 0.0, 5.0);
    myCamera->heightAngle = M_PI/2.0;
    myCamera->nearDistance = 2.0;
    myCamera->farDistance = 12.0;
    root->addChild(myCamera);

    SoCallback *myCallback = new SoCallback;
    myCallback->setCallback(myCallbackRoutine);
    root->addChild(myCallback);

    buildScene(root);

    // Render-Bereich und Fenster definieren
    SoXtRenderArea *myRenderArea
    = new SoXtRenderArea(myWindow);
    myRenderArea->setSceneGraph(root);
    myRenderArea->setTitle("OpenGL Callback");
    myRenderArea->setBackgroundColor(
    SbColor(.8, .8, .8));
    myRenderArea->show();

    SoXt::show(myWindow);

    // Inventor-Ereignisbearbeitung
    SoXt::mainLoop();
}
```

4. Vergleich von OpenGL und Java3D mit Inventor

OpenGL ist im Vergleich zu Inventor relativ low-level. Die API von OpenGL ist prozedural. Es versteht sich vor allem als hardwareunabhängige Render-Engine, weshalb z.B. auch Interaktionen kaum unterstützt werden (Ereignisse von Zeigegeräten sind hardwareabhängig und abhängig vom verwendeten Fenstersystem). Das wichtigste Ziel von OpenGL ist das gerenderte Bild.

Inventor basiert auf OpenGL und bietet über die Renderfähigkeiten von OpenGL hinaus direkte Unterstützung von Interaktionen und Animationen. Zusätzliche Komponenten für grafische Oberflächen erweitern die Fähigkeiten von Inventor zusätzlich. Bei Inventor steht nicht das gerenderte Bild im Vordergrund, vielmehr soll auch das interaktive Arbeiten dem visualisierten Modell unterstützt werden.

Obwohl OpenGL-Befehle in z.B. in Objekte bzw. Objekt-Methoden gekapselt werden können und eine komplexe Anwendung sicherlich nicht nur eine einzige Prozedur mit linear aufeinanderfolgenden OpenGL-Befehlen benutzt, um eine Szene zu rendern, sind aus Sicht von OpenGL OpenGL-Befehle linear angeordnet. Das erkennt man auch schon aus der linearen Struktur der *display lists*. Da nach dem Rendern keine strukturierten Informationen mehr über das Bild existieren, gibt es bei OpenGL im Allgemeinen keine Möglichkeit, die Szene nach dem Rendervorgang noch zu ändern. Die einzige Möglichkeit in OpenGL, Teile einer Grafik zwischenspeichern und bei Bedarf wiederzuverwenden, sind *display lists*, die jeweils eine lineare Abfolge von OpenGL-Befehlen enthalten und an beliebiger Stelle wieder abgespielt werden können. OpenGL bietet direkten Zugriff auf den *frame buffer*.

In Inventor ist die interne Struktur des Modells, nämlich der Szenengraph, das wesentliche Konzept. Szenen werden zunächst als Szenengraph konstruiert, und erst dann gerendert oder anderweitig weiterverarbeitet. Der Szenengraph kann zu jedem Zeitpunkt, d.h. auch nach dem Rendern, geändert werden. Dies ist auch die Grundlage für die einfache Implementierung von Interaktionen und Animationen in Inventor. Inventor bietet keinen direkten Zugriff auf den *frame buffer*.

Durch die höhere Abstraktionsebene von Inventor gegenüber OpenGL ist Inventor etwas langsamer. Jedoch sollte man beachten, dass Inventor intern sehr viele Optimierungen implizit macht und einige weitere Optimierungen unterstützt. Diese Optimierungen kann man bei Verwendung von OpenGL zwar grundsätzlich auch durchführen, jedoch ist die Programmierung dann recht aufwändig und erfordert sehr viel Wissen über die Optimierungsmöglichkeiten. Bei der Verwendung von

Inventor muss man also im Unterschied zu OpenGL praktisch kaum um Optimierungen kümmern.

Wie oben gezeigt, können Inventor und OpenGL relativ einfach kombiniert eingesetzt werden, sodass die Vorteile beider Bibliotheken innerhalb einer Anwendung ausgenutzt werden können.

Schaut man sich die Anwendungsgebiete der einzelnen Bibliotheken an, so fällt auf, dass OpenGL besser für Anwendungen geeignet ist, bei denen sich die Szene häufig ändert und bei denen Flexibilität und Performance im Vordergrund stehen, aber auf Interaktivität verzichtet werden kann. Dies trifft zum Beispiel für viele Simulationen zu, z.B. technische Simulationen oder Spiele. Inventor ist dagegen besser geeignet, wenn eine höhere semantische Ebene gewünscht wird und die Interaktivität und die Änderbarkeit der Szene im Vordergrund stehen. Beispiele dafür sind grafische Modellierungstools (z.B. CAD) oder Animationen. Aufgrund der höheren Abstraktionsebene von Inventor ist das Programmieren mit Inventor einfacher und vor allem schneller, da mit weniger Code-Zeilen das selbe erreicht werden kann.

Java3D ist, wie der Name schon sagt, ebenfalls eine 3D-Grafikbibliothek. Sie ist vollständig in Java geschrieben und bietet daher eine Java-API. In wesentlichen Punkten entsprechen sich Inventor und Java3D. Inventor ist wesentlich älter, Java3D ist erst in den letzten Jahren entwickelt worden. Das wichtigste Konzept von Inventor, der Szenengraph, ist bei Java3D das selbe, auch wenn der Szenengraph in Java3D etwas anders aufgebaut ist und die Terminologien etwas anders sind. Es gibt ebenfalls Engines für Animationen und Interaktionen werden ähnlich komfortabel unterstützt. Java3D gibt es für IRIX, HP-UX, Solaris, Linux und Windows. Java3D basiert wie Open Inventor auf lower level APIs. Es gibt eine Version, die auf DirectX basiert und eine, die auf OpenGL basiert.

5. Zusammenfassung

Open Inventor ist eine komplexe objektorientierte 3D-Bibliothek, die dem Stand der Technik entspricht. Sie setzt auf der anerkannt guten Bibliothek OpenGL auf und hat daher nahezu gleiche Render- und Performance-Eigenschaften wie OpenGL. Praktisch alle Features von OpenGL sind über Open Inventor nutzbar. Für Fälle, wo Open Inventor nicht so gut geeignet ist, sorgt OpenGL für die nötige Flexibilität.

Die Erstellung von 3D-Modellen mit Inventor ist intuitiv, da die Objekte des Modells auf der realen physikalischen Welt basieren.

Inventor ist erweiterbar. Es ist offen für Vererbung und mittels Callback-Knoten können flexibel Callback-Routinen getriggert werden.

Das Dateiformat entspricht nicht mehr ganz dem aktuellen Stand, XML wäre hier angebracht.

Inventor bietet eine flexible Ereignisverarbeitung. Der Anwendungsprogrammierer kann wählen zwischen Inventor-Ereignissen und Ereignissen des Fenstersystems.

Die in Inventor enthaltenen Objekte für Interaktionen mit der 3D-Szene sowie die Möglichkeit der Erweiterung von Inventor um weitere interaktive Objekte lassen praktisch keine Wünsche offen.

Zusätzlich bietet Inventor eine Reihe von Komponenten, mit denen man einfache Oberflächen erstellen kann.

Inventor ist daher das geeignete Werkzeug für die Erstellung von Tools zur grafischen Modellierung und für Animationen.

Inventor profitiert direkt von der Weiterentwicklung von OpenGL. Weiterentwicklungen von OpenGL können (falls die OpenGL-API nicht verändert wird) meist ohne Änderungen von Inventor genutzt werden.

Da die OpenGL-API von vielen Profi-Grafikkarten unterstützt wird, ist die Hardwareunterstützung von Inventor praktisch kaum ein Problem. Dies ist ein wichtiger Punkt für die Akzeptanz solcher Bibliotheken.

6. Referenzen

- [1] Wernecke, Josie. The Inventor Mentor: programming Object-oriented 3D graphics with Open Inventor, release 2. Addison-Wesley Publishing Company, 1994.
- [2] Open Inventor Architecture Group. Open Inventor C++ Reference Manual: The Official Reference Document for Open Inventor, Release 2. Addison-Wesley Publishing Company, 1994.
- [3] Wernecke, Josie. The Inventor Toolmaker: Extending Open Inventor, Release 2. Addison-Wesley Publishing Company, 1994.
- [4] web3D Consortium. Homepage der Extensible 3D (X3D) Graphics Working Group. <http://www.web3d.org/x3d.html>.
- [5] Foley, J.D., A. van Dam, S. Feiner, and J.F. Hughes, Computer Graphics Principles and Practice, 2d ed. Reading, Mass.: Addison-Wesley, 1990.
- [6] Neider, Jackie, Tom Davis, Mason Woo. OpenGL Programming Guide. Reading, Mass.: Addison-Wesley, 1993.
- [7] Newman, W., and R. Sproull, Principles of Interactive Computer Graphics, 2d edition. New York: McGraw-Hill, 1979.