# Aufgabe 2: Der Greifer

Martin Aumüller *
Universität zu Köln
basierend auf Aufgabe 2 von GraPA an der Universität Erlangen-Nürnberg,
`http://www9.informatik.uni-erlangen.de/GraPA/`

9. Juli 2013

## 1 Themen

In dieser Aufgabe werden folgende Themengebiete bearbeitet:

- Hierarchische Szenengraphen
- Speicherverwaltung mit Verweiszählung
- Selektion
- Ereignisse
- Animation
- Indizierte Geometriedaten

## 2 Einführung

In der zweiten Aufgabe des Praktikums zur Computergrafikprogrammierung lernst Du am Beispiel von OpenInventor/Coin3D die Grafikprogrammierung mithilfe von hierarchischen Szenengraphen. Inventor ist eine Objektbibliothek, die als Grundlage für das Darstellen wieder auf OpenGL zurückgreift.

Am Ende wird eine interaktive 3D-Grafik-Implentation des Brettspiels „Solitär" entstanden sein. Die Regeln von Solitär sind nicht kompliziert: Ein Spielstein wird auf dem Brett bewegt, indem er einen benachbarten Stein überspringt und dabei auf ein unbesetztes Feld gelangt. Der übersprungene Spielstein wird aus dem Spiel entfernt. Ziel des Spiels ist, möglichst viel Spielsteine zu entfernen, bevor kein Zug mehr möglich ist. Aber am besten probierst Du das Beispielprogramm aus, um das Resultat zu sehen!

Nach dem Start des Programms, kannst Du die Escape-Taste dazu benutzen, um zwischen dem Betrachtungs- und dem Interaktionsmodus hin- und herzuwechseln.

- Im Betrachtungsmodus verändert sich der Mauszeiger zu einer Hand. Dann kann man die Szene mit dem linken und mittleren Mausknopf drehen und verschieben.
- Im Interaktionsmodus wird der Mauszeiger als Pfeil dargestellt. In diesem Modus kannst Du Spielzüge vornehmen, indem Du einen Spielstein auf dem Brett anklickst und ihn durch Anklicken eines leeren Feldes dorthin versetzt – das alles ist natürlich nur im Rahmen der Regeln von Solitär möglich.

Die Grundfunktionalität des Spielfelds ist bereits im Gerüstprogramm vorhanden. Folgendes musst Du noch implementieren:

1. Stelle das Spielbrett mit Standard-Inventor-Knoten dar.
2. Erlaube die Auswahl von Spielfeldern, indem Du eine Behandlungsroutine für Selektionsereignisse implementiertst.
3. Benutze indizierte Geometriestrukturen, um den 6-eckigen Sockel des Greifarms zu modellieren.
4. Verwende Standard-Inventor-Knoten, um die hierarchische Struktur des Greifarms aufzubauen.
5. Füge Code zur Berechnung der Position des Greifarms hinzu.
6. Programmiere einen „idle sensor" zur Erzeugung von Animationsereignissen.
7. Behandle diese Ereignisse zum Durchführen der Animation.

## 3 Aufbau

Ziel dieser Aufgabe ist das Verstehen der Inventor-Programmierschnittstelle. Das Programm gliedert sich in zwei Teile, die miteinander interagieren:

- das Spielfeld, dargestellt durch die Klasse *Gameboard* und
- den Greifarm, implementiert in der Klasse *Grabber*.

In Tabelle 1 sind die Quelltextdateien und die darin zu findenden Klassen aufgeführt.

*aumueller@uni-koeln.de

| Datei | Klasse | Beschreibung |
|---|---|---|
| `main.cpp` | - | Programmstart |
| `Gameboard.cpp` | *Gameboard* | Solitär-Spielfeld |
| `Grabber.cpp` | *Grabber* | Greifarm |

Tabelle 1: Dateien und Klassen

# 4 Grundlegende Ideen bei Inventor

Bevor es losgeht, sehen wir uns einige grundlegenden Ideen, die die Inventor-Architektur beeinflusst haben, an. Inventor stellt umfassende Funktionalität zum Umgang mit der festverdrahteten OpenGL-Pipeline zur Verfügung und kann verschiedene Formen von Objekten mit Materialeigenschaften, Lichtquellen und die Kamerakonfiguration verarbeiten.

## 4.1 Szenengraphen

In Inventor werden Szenengraphen dazu benutzt, komplizierte 3D-Modelle hierarchisch aufzubauen. Diese Graphen bestehen aus Knoten, den sog. „nodes“, die die grundlegende Funktionalität auf objektorientierte Weise bereitstellen.

## 4.2 Knoten

Knoten („nodes“) sind die Objekte, die zusammengefügt werden, um einen Szenengraphen aufzubauen. Bevor wir die verschiedenen Arten von Knoten verwenden können, ist es notwendig, die Technik der Verweiszählung („reference counting“) zu verstehen. Das ist eine nicht nur im Umgang mit Inventor, sondern für objektorientierte Programmierung allgemein bedeutsame Technik.

### 4.2.1 Verweiszählung

Die grundlegende Idee hinter der objektorientierten Programmierung ist es, Objekte statt globaler Funktionen zu implementieren. Gewöhnlich werden Objekte mit dem *new*-Operator (der wiederum den Konstruktor der Klasse aufruft) erzeugt und durch Anwenden des *delete*-Operators (der den Destruktor aufruft) wieder gelöscht.

Sehen wir uns nun ein Programm an, das viele Objekte verwendet, wie z. B. Inventor. Einige Objekte erzeugen selbst mit *new* wieder neue Objekte und diese werden wieder von anderen Objekten verwendet. In dieser Situation ist es schwierig zu entscheiden, wer für das Löschen der Objekte verantwortlich sein soll, da durchaus mehrere Objekte Zeiger auf ein zu löschendes Objekt haben können. Eine Lösung für dieses Problem ist die Technik der Verweiszählung.

Verwendet man Verweiszählung, so legt man Objekte weiterhin unter Verwendung von *new* an, aber ruft nie explizit *delete* auf, um sie zu löschen. Stattdessen löscht sich das sich Objekt selbst, wenn sein Verweiszähler 0 wird. (Aber nicht automatisch, wenn er 0 ist – sonst würde sich jedes neuerzeugte Objekt automatisch sofort selbst löschen.) Das bedeutet, dass man dem Objekt mitteilen muss, wenn man einen Zeiger auf es behalten möchte. In Inventor geschieht das durch Aufruf der Methode *ref()*, die den Verweiszähler erhöht. Entsprechend hat man *unref()* aufzurufen, wenn man den Zeiger verwirft. Wenn das Objekt nun bemerkt, dass sein Verweiszähler wieder 0 erreicht hat, wird es für sich selbst *delete* aufrufen und seine Resourcen freigeben.

### 4.2.2 Generische Knoten – *SoNode*

Nachdem die Verweiszählung klar ist, können wir uns dem Aufbau hierarchischer Szenengraphen zuwenden. Inventor stellt eine Vielzahl von Knoten zur Verfügung. All diese Knoten sind von der gemeinsamen Basisklasse *SoNode* abgeleitet.

### 4.2.3 Gruppenknoten – *SoGroup* und *SoSeparator*

Um hierarchische Knotenstrukturen aufbauen zu können, benötigt man eine Möglichkeit zum Gruppieren mehrer Knoten. In Inventor sind solche Knoten von *SoGroup* abgeleitet. Für eine genauere Beschreibung sei ein Blick in die „man pages“ von *SoGroup* und *SoSeparator*, der beiden wohl wichtigsten Gruppenknoten, empfohlen.

- Ein *SoGroup*-Knoten gruppiert mehrere Kinderknoten. Die Kinderknoten werden sequentiell (in der Ordnung des Hinzufügens) vom Gruppenknoten durchlaufen. Kinderknoten können natürlich wieder selbst *SoGroup*- oder davon abgeleitete Knoten sein.

- *SoSeparator* ist von *SoGroup* abgeleitet. Im Unterschied zu *SoGroup* sichert dieser Knoten den OpenGL-Zustand ehe er seine Kinder durchläuft und stellt den alten Zustand danach wieder her (*glPush.../glPop...*). Diese Eigenschaft macht ihn zu einem wichtigen Helfer beim Aufbauen hierarchischer Szenengraphen.

### 4.2.4 Geometrieknoten – *SoShape*

Inventor stellt für ausgewählte Körper, so z. B. Quader, Kugeln, Kegel und Zylinder von *SoShape* abgeleitete Knoten zu ihrer Darstellung bereit - die „man pages“ zu *SoCube*, *SoSphere*, *SoCone* und *SoCylinder* enthalten die Details.

Will man allgemeinere Geometrie darstellen, so kommen die Knoten für indizierte Geometrie in Frage.

Beispielsweise kann man mit *SoIndexedFaceSet* unter Verwendung der Koordinaten aus einem *SoCoordinate3*-Knoten aus Polygonen bestehende Objekte rendern.

#### 4.2.5 Transformationsknoten

Für lineare Transformationen (Rotation, Translation und Skalierung) stellt Inventor von *SoTransformation* abgeleitete Knoten bereit. Es gibt entsprechend die Knoten *SoRotation*, *SoTranslation* und *SoScale* sowie *SoTransform*-Knoten, die eine Verkettung dieser Operationen durchführen.

#### 4.2.6 Materialknoten

Objektattribute wie Farbe und andere Oberflächeneigenschaften werden in Inventor durch eigene Knotentypen dargestellt. *SoMaterial* ist der wichtigste von ihnen.

Hier sei nochmals auf den Unterschied zwischen *SoGroup* und *SoSeparator* hingewiesen: verwendet man beispielsweise einen *SoMaterial*-Knoten unterhalb von *SoGroup*, so wirken sich dessen Materialeigenschaften auch auf die Objekte aus, die in übergeordneten Hierarchieebenen nach dem Materialknoten durchlaufen werden. Deshalb ist es oft einfacher, die Szenengraphen mit *SoSeparator*-Knoten aufzubauen. Die einführenden Kapitel von [2] erzählen mehr darüber.

# 5 Das Spielbrett

Im Gerüstprogramm sind bereits die Grundzüge des Spielbretts, einschließlich der Spielregeln, angelegt. Deine Aufgabe ist es, die grafische Darstellung des Spielbretts zu realisieren. Das Spielbrett sollte als ein Muster aus 3-dimensionalen Feldern (die auch eine Höhe besitzen) bestehen, so wie im Beispielprogramm. Die Felder sollten unterschiedlich eingefärbt sein, um ihre Grenzen deutlich zu machen. Hänge Deinen Programmtext zum Aufbau des Szenengraphen in die Methode *initSceneGraph()* ein.

## 5.1 Das Aufbauen des Spielbretts

Im vorbereiteten Code in `Gameboard.cpp` werden die Felder des Spielbretts durch ein $7 \times 7$-Ganzzahlfeld repräsentiert. Jedoch sind nicht alle Felder gültig. In *initGameboard()* werden die Felder vorinitialisiert. *initSceneGraph()* legt bereits einen Knoten vom Typ *SoSelection* – welcher von *SoGroup* abgeleitet ist – an. Dieser kann zur Implementation der Auswahl verwendet werden.

Obwohl es Euch überlassen ist, wie Ihr das Spielbrett realisiert, hier ein paar Tips, die nützlich sein könnten. Vielleicht ist es am einfachsten, für jedes Feld einen *SoSeparator*-Knoten anzulegen, unabhängig davon ob es gültig ist oder nicht. Für jedes gültige Feld sollte dieser Knoten folgende Kinder besitzen:

- einen *SoTransform*-Knoten, der das Feld an die richtige Stelle rückt,
- einen *SoMaterial*-Knoten zum Einfärben des Felds,
- einen *SoCube*-Knoten zum Erzeugen der Geometrie sowie
- natürlich den Spielstein, falls das Feld besetzt ist.

Hältst Du Dich an dieses Konzept, so kannst durch Anwenden von *getChild(i)* auf das Attribut *m_sceneGraph* einen Zeiger vom Typ *SoNode* auf das *i*-te Feld bekommen, den Du mit *static_cast* in einen vom Typ *SoSeparator* umwandeln kannst. Dieser besitzt dann natürlich nur für gültige Spielfelder Kinder.

## 5.2 Die Spielbrettfunktionalität

Damit das Spiel funktioniert, hast Du noch folgendes zu tun:

- Implementiere die Methode *removePiece()*, die die Geometrie des Spielsteins vom Spielfeld entfernt und den so erhaltenen Szenengraphen zurückgibt.
- Implementiere die Methode *insertPiece()*. Sie fügt den übergegebenen Teilgraphen einem Feld hinzu.
- Implementiere die Methode *getPositionOfPiece()*, die die Raumposition eines Spielsteins, d. h. den Ort, den der Greifarm anfahren muss, ermittelt.
- Implementiere eine Behandlungsroutine für Selektionsereignisse (siehe den folgenden Abschnitt).

## 5.3 Auswahl

Anders als reines OpenGL stellt Inventor eine einfache Schnittstelle zum Auswählen von Objekten zur Verfügung. Selektierbare Objekte müssen sich lediglich unterhalb eines *SoSelection*-Knoten befinden. Dann ist nur noch die Behandlungsroutine zu programmieren, die immer dann aufgerufen wird, wenn ein Unterobjekt des *SoSelection*-Knoten mit der Maus angeklickt wird.

In dieser Behandlungsroutine solltest Du auf folgende Art und Weise die Interaktion des Spielfelds mit dem Greifarm implementieren:

- Wird ein Spielstein (oder ein besetztes Feld) vom Spieler ausgewählt, dann sollte die Behandlungsroutine automatisch aufgerufen werden. Diese sollte dann die Methode *getPiece()* des Greifarms mit der Raumposition des Spielsteins aufrufen.

- Der Greifarm bewegt sich dann zu dieser Position und ruft die Methode *getPiece()* des Spielfelds auf und bekommt so den Teilgraphen des Spielsteins übertragen. Damit ist dieses Selektionsereignes abgearbeitet.
- Wählt der Benutzer dann ein anderes Spielfeld an, so muss die Behandlungsroutine die Regeln überprüfen und entscheiden, ob das zu einem erlaubten Spielzug führt. Falls ja, ist die Routine *setPiece()* des Greifarms mit der Raumposition des Zielfelds als Argument aufzurufen.
- Der Greifarm führt nun die Animation durch und bewegt sich zur angegebenen Position. Der Spielstein wird dann durch Aufruf der Spielbrett-Methode *setPiece()* dem Spielbrett zurückgegeben.

# 6 Der Greifarm

Du kannst es bestimmt schon nicht mehr erwarten, Solitär zu spielen. Aber wie die Spielsteine bewegen? Du lässt Dir die Arbeit von einem Greifarm abnehmen!

Der Greifarm ist ein virtueller Roboterarm, der sich ferngesteuert bewegt, die Spielsteine aufnimmt und an anderer Position wieder absetzt. Damit er jeden beliebigen Raumpunkt erreichen kann, benötigt der Arm drei unabhängige Bewegungsmöglichkeiten: entsprechend seines Aufbaus müssen Raumpunkte für ihn in Kugelkoordinaten adressiert werden.

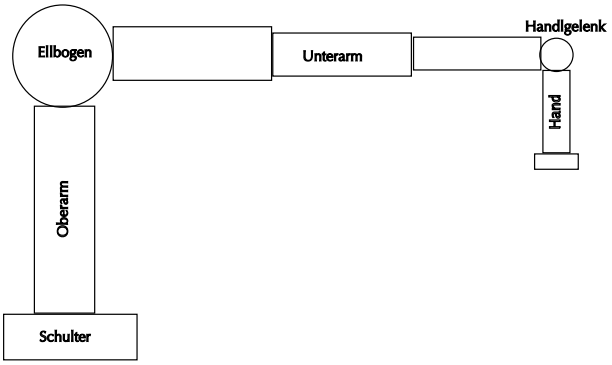


Abbildung 1: Aufbau des Greifarms

Der Greifarm muss folgende Freiheitsgrade besitzen:

- an der Schulter die Möglichkeit zur Rotation um die z-Achse,
- dazu senkrechte Rotationsmöglichkeit am Ellbogen und
- ein teleskopartiges Ein- und Ausfahren des Unterarms.

Zudem sollte die Hand am Handgelenk immer so orientert sein, dass sie senkrecht nach unten weist.

## 6.1 Aufbau des Greifarms

Hier im Detail, wie der Arm aufgebaut sein soll. Dies ist die Teile-Liste:

**Schulter** Das Fundament des Arms – seine Schulter – ist ein Prisma mit 6-eckiger Grundfläche. Es bleibt relativ zum Spielbrett immer in derselben Position. Erzeuge es mittels *SoIndexedFaceSet*, das auf die Koordinaten aus einem *SoCoordinate3*-Knoten verweist.

**Oberarm** Der Quader, der drehbar auf der Schulter steht, ist der Oberarm des Greifers.

**Ellbogen** Ein auf dem Oberarm liegender Zylinder ist der Ellbogen des Greifers. Durch Rotation um die Symmetrieachse des Zylinders, die immer parallel zur x-y-Ebene ist, wird die Höhe des Greifarms gesteuert.

**Unterarm** Drei teleskopartig ineinanderschiebbare Zylinder bilden den Unterarm des Greifers. Das ermöglicht die Veränderung des radialen Abstands des Fingers. Die drei Zylinder sollten eine gemeinsame Achse sowie gleiche Länge, aber unterschiedliche, in Richtung des Ellbogens zunehmende, Radien besitzen.

**Handgelenk** Das Handgelenk wird durch eine Kugel, die am Ende des dünnsten Zylinders angebracht ist, repräsentiert. Es bietet die nötige rotatorische Beweglichkeit, so dass die Hand immer in Richtung des Bodens zeigt.

**Hand** Unterhalb des Handgelenks befindet sich ein die Hand darstellender Kegel, dessen Achse immer senkrecht auf der x-y-Ebene steht.

**Finger** Der Finger wird von einem flachen Zylinder direkt unterhalb der Spitze des Handkegels gebildet, dessen Achse mit der der Hand übereinstimmt. Der Finger funktioniert wie ein Elektromagnet, der die Spielsteine anheben und abstellen kann.

Bitte beachtet: die einzelnen Teile sind starr und unveränderlich und können sich nicht verformen. Lediglich ihre Position und Lage zueinander verändert sich.

Der Greifarm sollte folgende Bewegungen vollführen können:

**Rotation** Abgesehen von der Schulter sollten alle Teile des Arms zusammen um die gemeinsame Symmetrieachse der Schulter und des Oberarms rotieren können.

**Abstand** Der Abstand zwischen Hand und Schulter kann durch gleichmäßiges Aus- und Einfahren der beiden kleineren Segmente des Teleskop-Unterarms variiert werden.

**Höhe** Der Abstand des Fingers zur Ebene des Spielbretts wird durch den Rotationswinkel am Unterarm gesteuert. Am Handgelenk muss entsprechend auch eine Rotation angewandt werden, damit die Hand immer senkrecht nach unten zeigt.

Das sollte genügen, um den Szenengraph durch Füllen der Methode *initSceneGraph()* aufbauen zu können. Um das gewünschte Ergebnis genau zu verstehen, ist es am besten, mit dem Beispielprogramm zu experimentieren.

### 6.2 Animation

Abschließend bleibt Dir nur noch, die Bewegung des Greifarms zu animieren. Die Bewegung sollte nicht durch eine *for*- oder ähnliche Schleife, sondern von Ereignissen gesteuert werden.

- Gib dem Greifarm ein Attribut vom Typ *SoIdleSensor*.
- Wann immer das Ablaufen der Animation nötig ist, stoße den *SoIdleSensor* mit *schedule()* an, so dass er – immer wenn das Programm nicht anderweitig beschäftigt ist, üblicherweise nach dem Rendern eines Frames – eine vorher festgelegte Behandlungsroutine aufruft.
- Implementiere eine solche Behandlungsroutine, die entsprechend dem aktuellen Zustand des Greifarms entscheidet, was zu tun ist. Die Animation sollte aus den folgenden Abschnitten bestehen:

  1. In der Bewegungsphase sind nur Rotation und die Ausfahrlänge des Unterarms inkrementell anzupassen, bis die Zielposition erreicht ist.
  2. Ist der Zielpunkt erreicht, sollte der Arm seine Hand senken und am Ende dieser Phase mittels *getPiece()*/*setPiece()* einen Spielstein aufnehmen oder absetzen.
  3. Wurde ein Spielstein aufgenommon oder abgesetzt, sollte sich der Arm wieder bis zu seiner Wartestellung heben. Ist diese erreicht, ist die Animation vollständig ausgeführt und der *SoIdleSensor* sollte nicht erneut angestoßen werden.

## 7 Zusammenfassung

Wenn Du bis hierher gekommen bist, solltest Du solides Wissen um die Verwendung der vorgefertigen Inventor-Knoten haben. Es gibt auch die Möglichkeit, eigene Knoten zu implementieren. Das beschreibt [3].

## 8 Bewertungsrichtlinien

Wenn Du die folgenden Bedingungen erfüllst, dann kannst Du die Höchstpunktzahl von 20 Punkten erreichen:

**1 Punkt** Der Quelltext ist vollständig in objektorientiertem C++ verfasst. Der Code kompiliert auf dem Referenzsystem mit dem GNU C++-Compiler ohne Fehler und erzeugt mit den Optionen `-Wall -O2` keine unnötigen Warnungen. Auch die anderen Programmierrichtlinien wurden befolgt.

**2 Punkte** Der gesamte Quelltext ist in englischer Sprache kommentiert und die Kommentare erklären die algorithmische Struktur des C++-Codes. Die Kommentare sind so formatiert, dass sie das Erzeugen einer HTML- und LaTeX-Dokumentation mit `doxygen` erlauben.

**2 Punkte** Der Szenengraph des Spielbretts ist vernünftig aufgebaut.

**1 Punkt** Die Positionen der Spielfelder und -steine werden korrekt berechnet.

**2 Punkte** Die Selektionsbehandlung funktioniert richtig und führt nicht zu unerwarteten Zuständen.

**2 Punkte** Das Übergeben von Objekten zwischen Greifer und Spielfeld ist korrekt und die Regeln der Verweiszählung werden beachtet.

**2 Punkt** Der Greifarm ist entsprechend den Anforderungen entworfen und sein Szenengraph sinnvoll aufgebaut.

**1 Punkt** Der 6-eckige Sockel des Greifarms wird richtig dargestellt.

**3 Punkte** Die Position des Greifarms wird richtig festgelegt.

**1 Punkt** Die Animation besteht aus verschiedenen Phasen.

**2 Punkte** Die Animation ist ereignisgesteuert und funktioniert richtig.

**1 Punkt** Man kann tatsächlich Solitär spielen.

## Literatur


[1] Doxygen manual. `http://www.stack.nl/~dimitri/doxygen/manual.html`, 2005.

[2] Open Inventor Architecture Group Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2*. Addison Wesley.

[3] Open Inventor Architecture Group Josie Wernecke. *The Inventor Toolmaker: Extending Open Inventor, Release 2*. Addison Wesley.

[4] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 3rd edition, 1997.