Projektarbeit
Computeranimierte Fertigungsprozesse für den virtuellen KVP-Workshop

Wagner Gilles

13. März 2006

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Problemstellung

Der kontinuierliche Verbesserungsprozess (KVP, Abb. 1.1) [1] beschreibt die regelmäßige methodische Feststellung von Problemen und Engpässen in der laufenden Produktion sowie die Ausarbeitung und Umsetzung von Verbesserungsvorschlägen zu deren Umsetzung [2]. Nachteile des KVP sind die aus dem Vorgehen resultierenden Stillstandszeiten der Produktion sowie in der Regel unzureichende oder fehlende Evaluierungsmöglichkeiten der Verbesserungsvorschläge vor deren Umsetzung in der Produktion. Mit Hilfe der Informationstechnologie können diese aus dem Ablauf des KVP resultierenden Nachteile beseitigt werden. Das Forschungsprojekt KoVir beschäftigt sich mit dieser Problemstellung. Gegenstand der Forschung ist die Reduzierung der Produktionsstillstände und die Verbesserung der Evaluierungsmöglichkeiten bei der Durchführung des KVP durch den Einsatz der Virtual Reality Technologie [3]. Virtual Reality ist eine Computer Technologie die dem Anwender die Möglichkeiten bietet, in eine

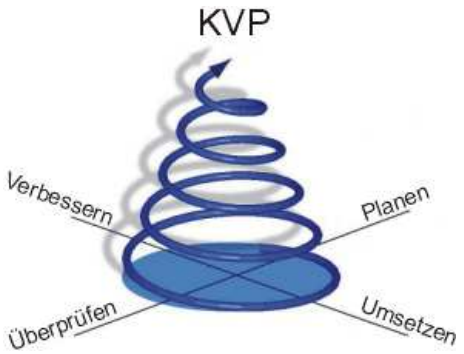

Abbildung 1.1: Der kontinuierliche Verbesserungsprozess

Abbildung 1.2: Die CAVE

virtuelle Umgebung einzutauchen (Immersion), diese in Echtzeit zu verändern (Interaktion) sowie dynamisches Verhalten darzustellen (Simulation) [4]. Das Forschungsprojekt zielt auf die Entwicklung eines Rahmenkonzepts für den virtuellen KVP. Es beinhaltet ein neuartiges VR-Modell und dessen prototypische Realisierung in einem System-Prototyp. Besondere Merkmale des VR-Modells sind die Integration unterschiedlicher Darstellungs- und Simulationsmethoden zur Verbesserung der Visualisierungs- und Simulationsmöglichkeiten.

1.2 Anwendungsszenario

Als Anwendungsszenario dient ein Teil der Produktionshalle der Firma John Deere in Zweibrücken. Hier werden Teleskoparme für Nutzfahrzeuge hergestellt. Dies geschieht in drei Arbeitstakten bei denen jeweils neue Einzelteile eingebaut werden. Der Produktionsgegenstand rollt dabei auf einem Transportwagen von einer Arbeitsstelle zur nächsten.

Als Virtual Reality Umgebung kommen eine CAVE (Abb. 1.2) und eine Powerwall zum Einsatz. Die CAVE (Cave Automated Virtual Environment) ist eine immersive Virtual Reality Technologie die für die Darstellung einer und Interaktion mit einer räumlichen Umgebung konstruiert ist. Eine CAVE ist im Allgemeinen ein Würfel mit bis zu sechs Projektionsflächen, die der Benutzer betreten kann. Die Powerwall ist eine Fläche auf die jeweils verschiedene Bilder für das rechte und linke Auge projeziert werden. Dies vermittelt den Eindruck räumlicher Tiefe.

1.3 Motivation und Überblick

Im Rahmen des KoVir Projekts werden Tools benötigt um große VRML-Datensätze von Produktionsprozessen effizient zu organisieren, Produktionsabläufe durch Computeranimation zu visualisieren und mögliche Verbesserungen rechnergestützt

zu erarbeiten. Die Tools arbeiten dabei mit einem eigenen Datenformat (kvp), als Schnittstelle zur VR-CAVE dient das VRML-Format [5].

In dieser Arbeit wird eine Möglichkeit vorgestellt große Datensätze zu modularisieren. Die Daten werden dabei durch Entfernung redundanter Information auf einen Bruchteil ihrer Ausgangsgröße reduziert. Weiterhin fördert diese Modularisierung die Übersichtlichkeit und organisiert die Daten effizient für eine komfortable Weiterverarbeitung.

Um ausgehend vom fertigen Produktionsgegenstand einen Fertigungsprozess zu simulieren benötigt man ein Tool das den Produktionsgegenstand in seine Einzelteile zerlegt. Zu diesem Zweck wurd ein kleines Tool entwickelt das hier ebenfalls vorgestellt wird.

Ausgehend von den so erzeugten Daten möchte man nun Produktionsabläufe simulieren. Dazu wurde ein Editor entwickelt der es ermöglicht KVP Szenarios zu laden und zu verändern oder gänzlich neue Szenarios zu erstellen. Den Produktionsprozess kann man anschließend als Animation darstellen und so Schwachstellen erkennen. Der Editor erlaubt es die Szenarios samt Animation abzuspeichern oder nach VRML zu exportieren für eine spätere Verwendung in der CAVE.

Kapitel 2

Konzept und Funktionalität

2.1 Datensätze modularisieren

2.1.1 Konzept

Die verfügbaren Geometriedaten der Fertigungsanlage liegen als CAD-Modelle in einer nach VRML konvertierbaren Datei (JT) vor. Dabei sind eigentlich identische Objekte, die aber an verschiedenen Positionen in der Halle stehn, als eigenständige Objekte abgespeichert (Abb. 2.2). Dieser Datensatz soll nun modularisiert werden. Dabei werden viele identische Objekte zu einem einzigen Objekt mit mehreren Instanzen zusammengefaßt. Von diesem wird dann der Objektschwerpunkt berechnet, und das Objekt so transformiert daß dieser Schwerpunkt mit dem Ursprung des Koordinatensystems zusammenfällt. Dann wird das so transformierte Objekt in eine eigene VRML-Datei abgespeichert. Gleichzeitig wird eine KVP-Masterdatei erzeugt welche im wesentlichen die vorherige Position eines jeden Objekts enthält und eine Referenz zu der Objektdatei. Insgesamt handelt sich also im Prinzip um eine Instanziierung der Objekte (Abb. 2.1).

2.1.2 Finden identischer Objekte

Wie erkennt man nun ob mehrere Objekte, bis auf ihre Position innerhalb der Halle identisch sind ? Eine recht aufwendige Methode wäre, alle Polygone miteinander zu vergleichen. Dies hat allerdings einige Nachteile. Erstens müßten die Objekt wirklich exakt dieselbe Triangulierung aufweisen. Zweitens wäre es wünschenswert, daß die Dreiecke für identische Objekte in derselben Reihenfolge abgespeichert sind. Ansonsten müßte man jedes Polygon eines Objekts mit allen Polygonen des andern Objekts vergleichen, was bei großen Datensätzen sehr aufwendig ist. Aufgrund der Konvertierung von JT nach VRML kann man nicht davon ausgehen daß diese beiden Bedingungen erfüllt sind. Daher kommen beim VRML Extractor, dem hier realisierten Tool, andere Verfahren zum Einsatz.

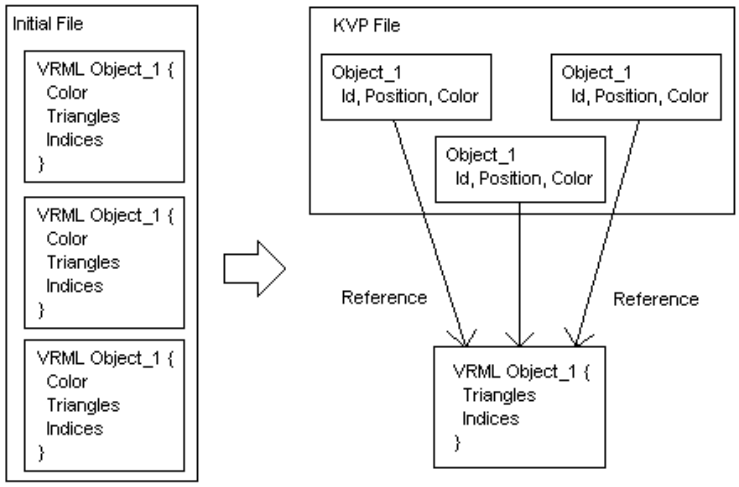

Abbildung 2.1: Modularisierungsprinzip

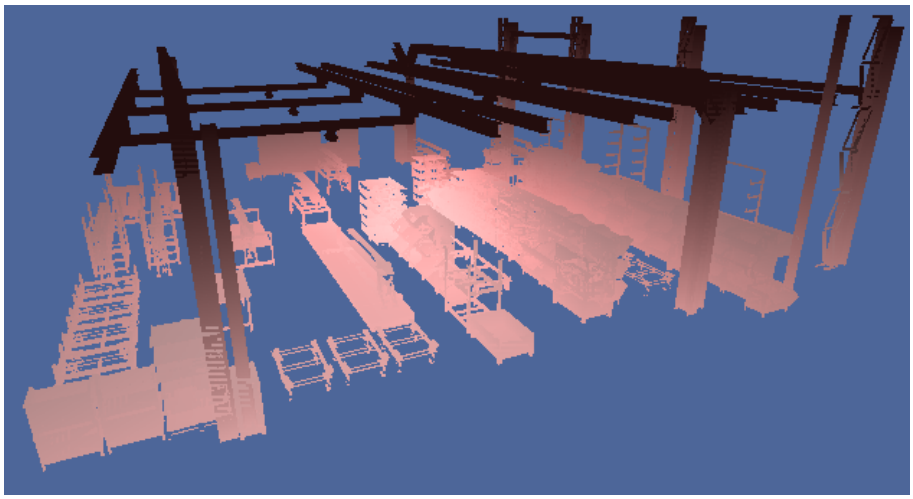

Abbildung 2.2: Ursprünglicher Datensatz

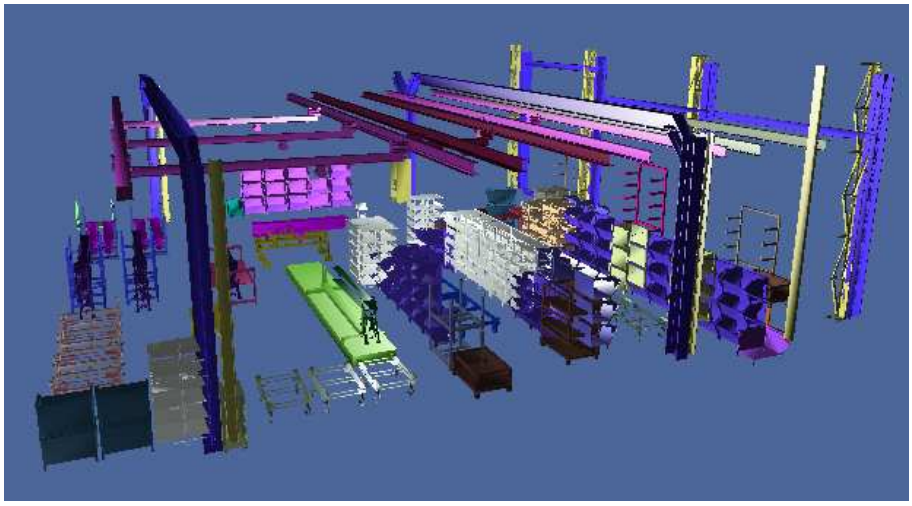

Abbildung 2.3: Modularisierte Daten - Gleiche Objekte haben dieselbe Farbe

Zunächst vergleicht man die Anzahl der Polygone. Je nach Datensatz liefert dies das gewünschte Ergebnis, aber die Gefahr zwei verschiedene Objekte als identisch abzuspeichern ist doch recht groß. Daher wird beim VRML Extractor zusätzlich für jedes Objekt eine Bounding Box erzeugt. Dazu werden die größten und kleinsten x, y und z Werte der Dreiecke ermittelt. Anschließend werden diese Bounding Boxes miteinander verglichen. Sind sie bis auf einen Toleranzwert (der Wert kann vom Benutzer eingestellt werden) identisch und stimmt zusätzlich die Polygonanzahl überein, betrachtet der VRML Extractor die Objekte als identisch. Ein zusätzlicher Vorteil bei dem Verfahren ist, daß auf einmal erzeugte Objektdateien nicht mehr zugegriffen werden muß. Das Programm merkt sich einfach die Bounding Box und Polygonzahl und kann damit den Test durchführen. Für Objekte die mit keinem bisherigen (Bounding Box, Polygonanzahl)-Paar übereinstimmen, wird eine neue Datei erzeugt und ein neues Paar zur Liste hinzugefügt. Während der Modularisierung generiert das Programm eine zufällige Farbe für jedes Objekt. Identische Objekte haben also später dieselbe Farbe (Abb 2.3). Damit kriegt man leicht einen Überblick ob das erzielte Resultat zufriedenstellend ist.

Je nach Datensatz besteht die Möglichkeit daß gleiche Objekte unterschiedlich orientiert sind. In diesem Fall könnte man die Objekte noch zusätzlich rotieren, bevor man überprüft ob sie identisch sind. Beim John Deere Datensatz ist dies jedoch nicht notwendig.

2.1.3 Objektursprung ermitteln

Um einen Objektschwerpunkt $\overline{P}$ zu berechnen wird folgendes Verfahren angewandt: Von den Eckpunkten aller Polygone werden jeweils die x, y und z Werte

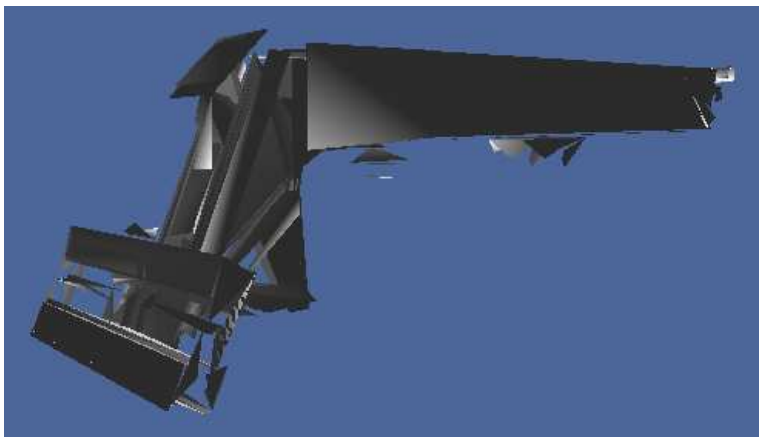

Abbildung 2.4: Ausgangsobjekt

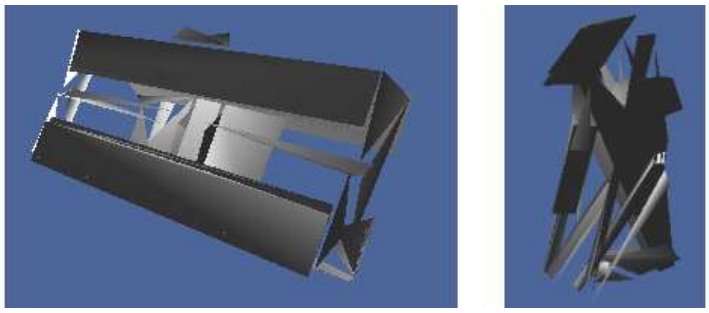

Abbildung 2.5: Extrahierte Objektteile

zusammengerechnet, und dann durch die Anzahl der Polygone dividiert:

$$\overline{P} = \frac{1}{n} \sum_{i=1}^{n} P_i$$

Der so ermittelte Punkt wird als Ursprung eines lokalen Koordinatensystems verwendet. Die Koordinaten der Punkte des Objekts werden in diesem Koordinatensystem neu ermittelt und in eine eigene VRML Datei abgespeichert. Gleichzeitig dient der Punkt als Referenzpunkt für die spätere Positionierung innerhalb der Szene. Der Punkt wird daher auch als Translationsvektor in der KVP Masterdatei abgespeichert.

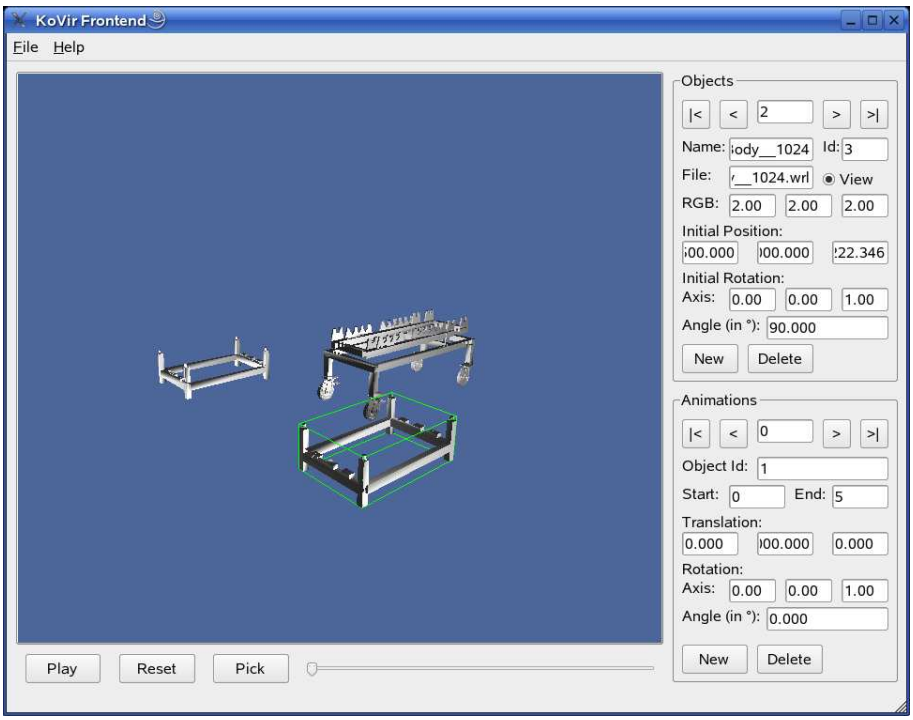

Abbildung 2.6: KoVir Frontend

2.2 Einzelteile extrahieren

Um die Montage des Teleskoparms (Abb. 2.4) zu simulieren, müssen wir diesen zuerst in Einzelteile (Abb. 2.5) zerlegen. Dazu benötigen wir ein Tool mit dem wir die Polygone einzelner Bereiche aus dem Objekt ausschneiden können. Diesen Zweck erfüllt das Programm Polygon Extractor. Es arbeitet nach einem simplen Prinzip. Als Eingabeparameter erwartet das Programm zwei Punkte welche die Eckpunkte eines Würfels ergeben. Danach werden alle Polygone die innerhalb dieses Würfels liegen, samt Normalen und neuer Indizierung in eine eigene Datei abgespeichert. Die Funktionalität dieses Programms ist eingeschränkt, als schnell implementierbare Lösung für unsere Zwecke aber ausreichend.

2.3 Animationseditor

2.3.1 Überblick

Um den Produktionsprozess simulieren zu können benötigen wir ein Tool mit dem man die einzelnen Arbeitsschritte als Computeranimation darstellen und bearbeiten kann. Die Bearbeitung soll dabei über eine graphische Benutzero-

berfläche möglich sein. Zur Sicherung der Daten soll ein intuitiv verständliches Masterdatei-Format entwickelt werden. Masterdatei bedeutet, daß in diesen Files nur die wichtigsten Eigenschaften der Objekte und Animationen definiert sind; insbesondere sollen die Masterdateien keine Geometriedaten enthalten. Diese stehen in VRML Dateien für die in der Masterdatei jeweils eine Referenz angegeben wird. Dieses Konzept garantiert Übersichtlichkeit und eine leichte Handhabung.

Das KoVir Frontend Tool (Abb. 2.6) bietet die Lösung für diese Aufgaben. Es ist ein Editor für die Erstellung computeranimierter Fertigungssequenzen basierend auf den extrahierten Daten. Das Programm erlaubt es einzelne VRML Objekte oder ganze KVP Szenarios zu laden und anzuzeigen. Diese kann man beliebig durch Hinzufügen oder Löschen einzelner Objekte verändern. Man kann auch völlig neue Produktionshallen aus einzelnen VRML Objekten zusammenstellen. Für die so erstellten Szenen kann man dann Animationen erzeugen und abspielen, um laufende Produktionsprozesse zu simulieren. Man kann diese Szenarios im KVP Format abspeichern oder zu einer animierten VRML Datei exportieren, was insbesondere für ein späteres Abspielen auf der Powerwall oder in der CAVE von Interesse ist.

2.3.2 KVP und VRML Format

Die Masterdatei für die KVP-Daten enthält keine Geometriedaten, sondern nur Referenzen zu Dateien im VRML Format, in denen die Triangulierung der Objekte steht. Auf diese Weise bleibt die Übersichtlichkeit erhalten. Eine detaillierte Beschreibung des VRML Formats enthält die VRML97 Spezifikation [5]. Die für diese Anwendung benutzten VRML Daten bestehen im Wesentlichen aus Dreiecken, deren Normalen und einer entsprechenden Indizierung der Daten. Hier ein kleines Beispiel für die verwendeten VRML Daten:

```
#VRML V2.0 utf8.
#Generated by VRMLExtractor.

DEF Body__1024 Transform {
children [
Shape {
geometry IndexedFaceSet {
coord Coordinate { point [
  -539 340 -101, -539 340 -181, -599 340 -101,
  -599 340 -101, -539 340 -181, -599 340 -181,
  -599 -48 -61, -599 -48 -101, -539 -48 -61,
  ...
] }
normal Normal { vector [
  0 1 0,  0 1 0,  0 1 0,
  0 1 0,  0 1 0,  0 1 0,
  0 -1 0,  0 -1 0,  0 -1 0,
  ...
] }
```

```
coordIndex [
  0, 1, 2, -1,
  3, 4, 5, -1,
  6, 7, 8, -1,
  ...
]
normalIndex [
  0, 1, 2, -1,
  3, 4, 5, -1,
  6, 7, 8, -1,
  ...
] } } ] }
```

Das KVP Format ist in zwei Teile aufgeteilt. Im ersten Teil, dessen Beginn durch das Schlüsselwort 'static' gekennzeichnet ist, steht die statische Beschreibung der Szene. Hier werden die Objekte der Szene aufgelistet, samt folgender Informationen: Objektname, Objekt Id, Referenz zu einer VRML Datei, Position und Orientierung innerhalb der Szene, sowie die Farbe des Objekts und ein boolescher Wert der angibt ob das Objekt angezeigt werden soll, oder nicht. Im zweiten Teil der KVP Datei, sind die Animationen abgespeichert. Dieser Teil beginnt mit dem Schlüsselwort 'animation'. Darauf folgt eine Liste mit Animationen. Die Angaben bestehen aus der Id des zu bewegenden Objekts, einer Start- und einer Endzeit, sowie aus einer Translation und/oder einer Rotation. Translationen werden durch einen Vektor definiert, Rotationen durch einen Vektor, der die Richtung der Rotationsachse durch den Objektschwerpunkt angibt und einen Winkel (in Grad) um den gedreht werden soll. Nachfolgend ein Beispiel für eine Datei im KVP Format:

```
# KVP File. Generated by KoVir Frontend.

static
  o:Body__1024
    i:2
    p:true
    f:data_mm/Body__1024.wrl
    r:0.000000,0.000000,1.000000,90.000000
    t:1500.147000,2009.953000,222.346000
    c:2.000000,2.000000,2.000000
  o:Body__1024
    i:3
    p:true
    f:data_mm/Body__1024.wrl
    r:0.000000,0.000000,1.000000,90.000000
    t:-1500.000000,2000.000000,222.346000
    c:2.000000,2.000000,2.000000
  ...
```

```
animation
  i:2
    s:0
    e:5
    t:0.000000,2000.000000,0.000000
  i:3
    s:5
    e:10
    r:0.000000,0.000000,1.000000,90.000000
    t:1500.000000,2000.000000,0.000000
  ...
```

Zum Einlesen der Formate benötigt man zwei Parser, zum Speichern und Exportieren sind Ausgaberoutinen notwendig.

2.3.3 KVP und VRML Parser

Der KVP- und der VRML-Parser zum Einlesen von Datensätzen aus Dateien funktionieren nach dem selben Prinzip. Die Dateien werden geöffnet und Zeile für Zeile abgearbeitet. Dabei wird nach bestimmten für das jeweile Format charakteristischen Stichwörtern gesucht. Wurde ein Stichwort gefunden so springt das Programm in eine Verzweigung und liest die dahinter stehenden Daten ein. Diese Daten werden dann abhängig vom Stichwort interpretiert, im Falle einer VRML Datei z.B. als Dreiecke, Normalen, Indizes, ... im Falle einer KVP Datei z.B. als Rotation, Translation, Farbe, u.s.w. Finden die Parser die Stichwörter 'o:' bei KVP oder 'DEF' bei VRML, so beginnt die Definition eines neuen Objekts. Deshalb werden alle vorher eingelesenen Daten zuerst mit dem alten Objekt verknüpft, bevor das Einlesen des neuen Objekts beginnt. Gleiches gilt für das Stichwort 'i:' im Animationsabschnitt der KVP Datei.

2.3.4 Die Szene anzeigen

Zur Darstellung der 3D Szene wird die OpenGL Schnittstelle [6] benutzt. Die einzelnen Objekte werden in CallLists gespeichert, welche durch die Objekt-Id abgerufen werden können. Beim Zeichnen der Objekte in die CallList wird eine Matrix auf die Punkte des Objekts multipliziert, die der korrekten Position und Orientierung des Objekts in der Szene entspricht. Läuft keine Animation so enthält diese Matrix die Anfangsposition und -orientierung. Läuft eine Animation so enthält die Matrix die richtige Position und Orientierung für den aktuell anzuzeigenden Frame. Wie diese Matrizen ermittelt werden, wird in Abschnitt 2.3.6 erklärt.

Die Kamera gibt an wo im Raum der Bildschirm liegt, auf den projeziert wird. Verändert man die Position der Kamera, so wird aber in der Praxis nicht die Projektionsebene neu ausgerichtet, sondern die gesamte Szene nochmal so transformiert, daß ihre Projektion in der gewünschten Position und dem

gewünschten Winkel auf den Bildschirm fällt. Mathematisch gesehn ist die Kamera also nichts anderes als eine weitere Transformation der Gesamtszene. Daher wird die Kamera auch in Form einer ModelView Transformationsmatrix angegeben. Diese gibt die relative Position von Kamera und Szene an. In unserm Editor ist die Kamera frei mit der Maus per Trackball positionierbar. Die Positionierung erfolgt dabei wie bei Objekten durch eine Kombination aus Translationen und Rotationen. Siehe hierzu ebenfalls Abschnitt 2.3.6.

Die Abbildung auf den Bildschirm erfolgt mittels einer perspektivischen Projektion. Dies geschieht bei OpenGL mittels der Funktion: void gluPerspective (GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far). Der Parameter fovy gibt dabei den Blickwinkel in der yz-Ebene an, aspect das Seitenverhältnis (x/y) der Projektionsebene, near und far den Abstand des Augpunkts zur 'near clipping plane' und zur 'far clipping plane' an. Siehe dazu auch Abb. 2.7. Eine perspektivische Projektion dieser Form ist mathematisch gesehn nichts anderes als eine Multiplikation der Objektdreiecke mit einer Projektionsmatrix der Form

$$P = \begin{pmatrix} \frac{cot(fovy/2)}{aspect} & 0 & 0 & 0 \\ 0 & cot(fovy/2) & 0 & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{2*far*near}{near-far} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Eine detaillierte Beschreibung des OpenGL Kamera- und Projektionsmodells findet man in der OpenGL Referenz [6]. Die Projektion auf die Ebene $(x', y', 0, w)^T$ eines Punktes $(x, y, z, w)^T$ im 3D-Raum errechnet man also durch folgende Transformations-Pipeline:

$$\begin{pmatrix} x' \\ y' \\ 0 \\ w \end{pmatrix} = P \times C \times M \times \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

wobei P die Projektionsmatrix, C die ModelView Matrix (Kamera) und M die Transformationsmatrix des Objekts ist.

2.3.5 Der Trackball

Der Trackball ermöglicht zoomen, translatieren und drehen der Kamera um den Szenenmittelpunkt. Der Szenenmittelpunkt wird beim Laden oder Erstellen der Szene ermittelt. Um die Kamera nun zu bewegen klickt man mit einer der drei Maustasten in den Viewport: linke Maustaste zum Drehen, mittlere Maustaste zum translatieren und rechte Maustaste zum Zoomen. Die Position des Mauszeigers wird nun ermittelt und zwischengespeichert. Bewegt man die Maus, so wird die neue Position des Mauszeigers festgestellt. Dann wird proportional zu der Bewegung (neue Position - alte Position) gedreht, translatiert oder gezoomt. Dazu wird die OpenGL Modelview Matrix verändert. Zum Translatieren wird

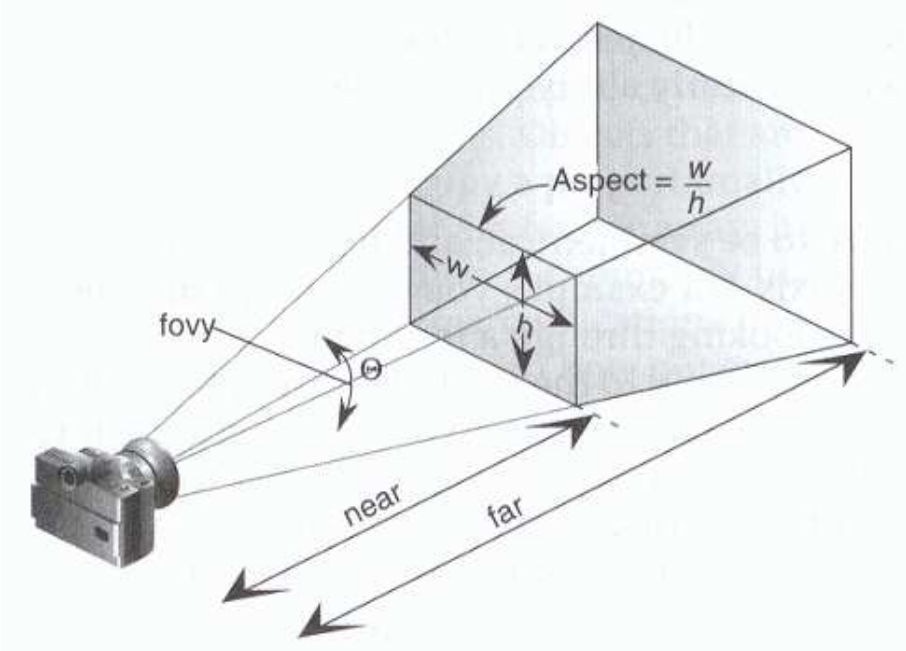

Abbildung 2.7: Perspektivische Projektion

eine Matrix erstellt mit Translationsvektor

$$(c * dx * scale, c * dy * scale, 0)^T$$

und diese auf die Modelview Matrix multipliziert. c ist dabei eine Konstante, die man so wählt dass man die gewünschte Geschwindigkeit erreicht. dx und dy sind die Differenz zwischen alter und neuer Position, scale ist die Größe der Szene.

Zum Drehen der Szene werde zuerst zwei Rotationsachsen bestimmt, abhängig von der aktuellen Modelview Matrix. Diese Rotationsachsen sollen der x- und der y-Achse des Bildschirms entsprechen. Deren Richtungsvektoren im globalen Koordinatensystem erhält man in dem man die ersten zwei Zeilen der ModelView Matrix ausliest und normiert. Dann wird die Modelview Matrix zum Ursprung des globalen Koordinatensystems translatiert. Die Kamera wird proportional zu dx um die y-Achse und proportional zu dy um die x-Achse gedreht. Danach wird sie wieder an ihren Ausgangsort zurücktranslatiert.

Beim Zoomen handelt es sich einfach um eine zu dy proportionale Translation der Kamera in die Richtung in die die Kamera gerade gerichtet ist, also entgegengesetzt der Richtung der virtuellen z-Achse des Bildschirms. Den Richtungsvektor der z-Achse im globalen Koordinatensystem erhält man durch auslesen der dritten Zeile der Modelview Matrix.

2.3.6 Transformationen im Raum

Um die Szene korrekt anzuzeigen und Animationen abzuspielen benötigt man ein Verfahren um Objekte und die Kamera im Raum zu drehen und zu verschieben. Diese Transformationen werden in OpenGL mit mit homogenen 4×4 Matrizen durchgeführt. Transformationen werden immer zusammengesetzt aus Translationen und Rotation. Man kann davon beliebig viele hintereinander ausführen. Wichtig ist dabei, daß man beachtet, daß die Matrizenmultiplikation nicht kommutativ ist. Das heißt die Reihenfolge bei der Multiplikation ist nicht beliebig. $A \times B$ ist nicht unbedingt gleich $B \times A$. Eine Translation um einen Vektor $(d, e, f)^T$ ergibt folgende homogene Matrix T:

$$\begin{pmatrix} 1 & 0 & 0 & d \\ 0 & 1 & 0 & e \\ 0 & 0 & 1 & f \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Eine Rotation um die durch den Objekturspung gehende Achse mit dem Vektor $(a, b, c)^T$ und Winkel w ergibt folgende homogene Matrix R:

$$\begin{pmatrix} cos(w) + a^2(1 - cos(w)) & ab(1 - cos(w)) + sin(w)c & ac(1 - cos(w)) - sin(w)b & 0 \\ ab(1 - cos(w)) - sin(w)c & cos(w) + b^2(1 - cos(w)) & bc(1 - cos(w)) + sin(w)a & 0 \\ ac(1 - cos(w)) + sin(w)b & bc(1 - cos(w)) - sin(w)a & cos(w) + c^2(1 - cos(w)) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Der Vorteil von homogenen Matrizen liegt darin, dass man alle notwendigen Transformationen als Matrizenmultiplikation darstellen kann. Bei nicht-homogenen

Matrizen wäre eine Translation nur dadurch machbar, daß man eine Addition des Translationsvektors durchführt. Da die Matrizenmultiplikation aber durch gängige Hardware speziell unterstützt wird, ist diese schneller als die Addition. Eine Kombination aus Translation und Rotation erhält man durch Matrizen der Form:

$$A = T \times R =$$

$$\begin{pmatrix} cos(w) + a^2(1 - cos(w)) & ab(1 - cos(w)) + sin(w)c & ac(1 - cos(w)) - sin(w)b & d \\ ab(1 - cos(w)) - sin(w)c & cos(w) + b^2(1 - cos(w)) & bc(1 - cos(w)) + sin(w)a & e \\ ac(1 - cos(w)) + sin(w)b & bc(1 - cos(w)) - sin(w)a & cos(w) + c^2(1 - cos(w)) & f \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Um Punkte durch solche Matrizen zu transformieren führt man eine einfache Matrix-Vektor Multiplikation durch:

$$P' = A \times P$$

Um i Transformationen hintereinander auszuführen multipliziert man i solcher Matrizen:

$$A = A_i \times A_{i-1} \times ... \times A_2 \times A_1$$

Für Animationen wird nun folgendes gemacht: Mit jedem Objekt der Szene ist eine Matrix M_{t_i} verbunden die seine Position und Orientierung zum aktuellen Zeitpunkt t_i enthält. Um die Position und Orientierung des Objekts im nächsten Frame zu berechnen wird eine Matrix A obiger Form berechnet. Diese ist abhängig von der seit dem letzten Frame verstrichenen Zeit. Diese neue Matrix wird dann auf die alte Objektmatrix multipliziert.

$$M(t_{i+1}) = A(t_{i+1} - t_i) \times M(t_i)$$

Prinzipiell ist eine Bewegung also nur eine Hintereinanderausführung vieler solcher Matrixmultiplikationen. Für jedes Zeitintervall $I_i = [t_i, t_{i+1}]$ berechnen wir eine Matrix A_{i+1}. Die Position und Orientierung eines Objekts zum Zeitpunkt t_m errechnet sich dann durch:

$$M(t_m) = A_m(t_m - t_{m-1}) \times A_{m-1}(t_{m-1} - t_{m-2}) \times ... \times A_1(t_1 - t_0) \times M(t_0)$$

wobei $M(t_0)$ die Anfangsposition und -orientierung des Objekts in der Szene angibt.

2.3.7 Objekte manipulieren

Um Objekte in der Szene manipulieren zu können, benötigt man die Möglichkeit ein Objekt auszuwählen. Die GUI bietet die Möglichkeit die Objekte in einer Liste direkt zu durchlaufen. Bei großen Szenen mit vielen Objekten ist dies jedoch sehr umständlich. Daher wird ein Verfahren benötigt um Objekte per Mausklick in die Szene anzuwählen. Hierzu rendert man um die Koordinaten des Mausklicks herum eine kleine Fläche von z.B. 10×10 Pixeln. Objekte deren Projektion diesen Bereich trifft kommen zur Auswahl in Frage. Im Allgemeinen wird der Benutzer vermutlich das Objekt auswählen wollen, das am nächsten

bei der Kamera liegt. Die Parameter des ausgewählten Objekts werden dann in der GUI angezeigt. Diese kann man nun manipulieren um Name, Id, Position, Orientierung oder Farbe des Objekts zu verändern. Zusätzlich wird um das gewählte Objekt herum ein Würfel gezeichnet. Dieser Würfel entspricht einfach den Koordinaten einer Bounding Box für das Objekt. Die Bounding Box wird beim Laden der Objekte ermittelt, indem man nach den kleinsten und größten x,y und z Koordinaten der Dreiecke sucht. Für jede Transformation die das Objekt betrifft muß die Bounding Box ebenfalls translatiert und/oder rotiert werden. Die OpenGL Schnittstelle unterstützt dieses Verfahren speziell, Details dazu in der OpenGL Referenz [6].

2.3.8 Animationen erstellen

Neue Animationssequenzen erstellt man durch manuelles Zufügen von Animationsabschnitten in der KVP Datei oder mit der GUI. Mit dem 'New' Button wird eine neue Animation in der Liste angelegt. Hier gibt man nun die Objekt Id des zu animierenden Objekts an. Dann gibt man Startzeitpunkt und Endzeitpunkt an, sowie eine Translation und/oder Rotation. Durch das Ausführen mehrer solcher Transformationen nacheinander kann man jede benötigte Animation erstellen. Das Löschen von Animationen ist per 'Delete' Button möglich. Vergleiche Abb. 2.6.

2.3.9 Animationen abspielen

Hat man alle nötigen Animationen erstellt, kann man sie per 'Play' Button abspielen. Das Programm ruft dann die Funktion 'startAnimation' auf. Diese ermittelt zunächst den Gesamtstart- und Gesamtendzeitpunkt aller Animationen. Daraus errechnet sich die Animationsdauer. Nun wird noch die Systemzeit ein erstes Mal ausgelesen um den Startzeitpunkt der Animationsausgabe festzuhalten. Um einen weiteren kleinen Performance Gewinn zu erhalten, kann man die Animationen noch nach Startzeit sortieren. Dies hat den Vorteil daß man nicht alle Animationen in der Schleife durchlaufen muß, sondern abbrechen kann, sobald die Startzeit größer ist als die aktuelle Zeit während der Animation.

```
20  film_start = SMALLEST anim.start
30  film_end = LARGEST anim.end
40  film_length = film_end - film_start
50  start_time = CLOCK
60  curr_time = CLOCK - start_time
70
80  WHILE (curr_time < film_length)
90    prev_time = curr_time
100   curr_time = CLOCK - start_time
110   FOR (ALL anim)
120     IF (curr_time LIES BETWEEN anim.start AND anim.end)
130       length = anim.end - anim.start
```

```
140       TRANSLATE object TO anim.trans*(curr_time-prev_time)/length
150       ROTATE object TO anim.angle*(curr_time-prev_time)/length
160       DRAW object
170     END IF
180   END FOR
190 END WHILE
```

Jetzt beginnt die eigentliche Animationsausgabe. Die Funktion versucht nun soviele Frames wie möglich darzustellen, für eine möglichst geschmeidige Animation ohne Ruckel-effekte. Vor jedem Frame wird die Systemzeit vom letzten Frame zwischengespeichert. Dann wird die aktuelle Systemzeit ermittelt. Aus der Differenz ergibt sich die zwischen dem letztem und dem aktuellen Frame verstrichene Zeit. Proportional zu dieser verstrichenen Zeit werden die Objekte für den aktuellen Frame translatiert und/oder rotiert. Objekte werden nur transformiert, wenn für sie eine Animation zur aktuellen Animationszeit definiert ist. Durch die Transformationen proportional zur verstrichenen Zeit zwischen den Frames wird die Geschwindigkeit der Transformationen konstant gehalten. Dies ermöglicht völlig gleichmäßige Animationsabläufe, selbst wenn das System durch andere Tasks ungleichmäßig belastet wird. Die Schleife zur Animationsausgabe wird solange durchlaufen, bis die Filmdauer erreicht ist.

Zur Transformation der Objekte wird erst die alte CallList gelöscht. Dann wird eine neue Matrix ermittelt, die der neuen Position und Orientierung entspricht. Dazu werden Translationsmatrix und Rotationsmatrix entsprechend der verstrichenen Zeit berechnet und nacheinander auf die alte Objektmatrix multipliziert.

$$M_{t_{i+1}} = T(t_{i+1} - t_i) \times R(t_{i+1} - t_i) \times M_{t_i}$$

Jetzt wird das Objekt neu gezeichnet mit Position und Orientierung der so ermittelten Matrix und in einer CallList abgespeichert.

Beim letzten Schritt einer Animation kann man noch zusätzlich überprüfen wie weit bisher bewegt wurde, und das Objekt dann im letzten Schritt nur noch um die Differenz zwischen der gewünschten Gesamttransformation und der bisherigen Transformation bewegen. Dadurch vermeidet man die Akkumulierung von Rundungsfehlern. Die mathematischen Grundlagen zu den Transformationen findet man in Abschnitt 2.3.5. Die Abbildungen 2.8 bis 2.10 zeigen eine mit dem KoVir Frontend erstellte Animation.

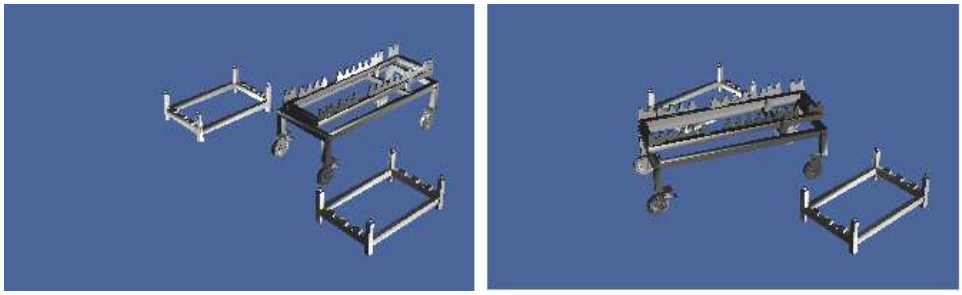

Abbildung 2.8: Animation eines Rollwagens

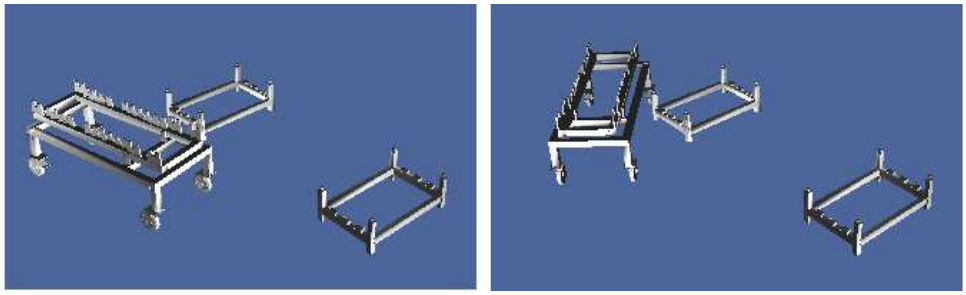

Abbildung 2.9: Er fährt in einer kreisförmigen Bewegung um die Kästen

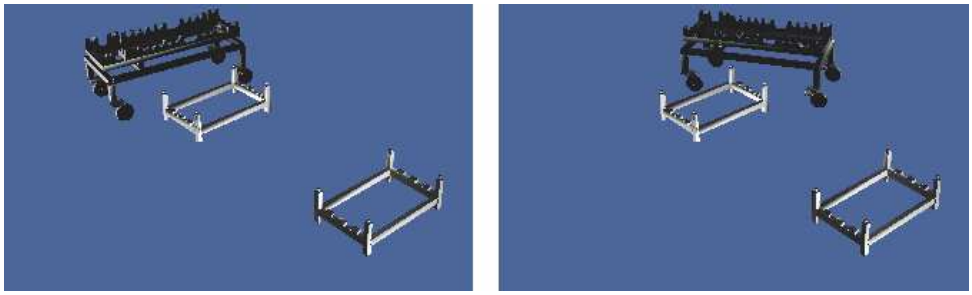

Abbildung 2.10: Dazu wird er gleichzeitig rotiert und translatiert

2.3.10 Animationen exportieren

Zusätzlich zum Abspeichern der Szene samt Animationen im KVP Format, gibt es die Möglichkeit ins VRML Format zu exportieren. Dies wird benötigt um die Szene in die Cave zu laden. Die erzeugte animierte VRML Datei ist wie folgt aufgebaut: Für jedes Objekt wird eine Abschnitt erzeugt der Form:

```
DEF ObjektName TRANSFORM {
CHILDREN [
INLINE { URL DateiName } ]
TRANSLATION a b c   // (a,b,c) Translationsvektor
ROTATION a b c w }   // (a,b,c) Rotationsvektor, w Winkel
```

Dann wird ein Zeitsensor definiert, welcher vom Filmanfang bis zum Filmende läuft, und dieses Intervall in einer Endlosschleife wiederholt.

```
DEF Uhr TIMESENSOR {
CYCLEINTERVAL film_dauer
LOOP TRUE }
```

Für jede Translation wird ein Positionsinterpolator erzeugt mit Startzeit, Endzeit (key), sowie mit dem Translationsvektor (keyValue). Ausserdem wird der Interpolator mit dem Zeitsensor verknüpft.

```
DEF Trans_Interpolator_ObjektId POSITIONINTERPOLATOR {
KEY [ startzeit endzeit ]
KEYVALUE [ 0 0 0, a b c ] }   // (a,b,c) Translationsvektor
ROUTE Uhr.FRACTION_CHANGED TO TransInt_ObjektId.SET_FRACTION
ROUTE TransInt_ObjektId.VALUE_CHANGED TO ObjektName.SET_TRANSLATION
```

Für jede Rotation wird ein Orientationinterpolator erzeugt mit Startzeit, Endzeit (key), sowie mit dem Rotationsvektor und -winkel(keyValue). Der Interpolator wird wiederum mit dem Zeitsensor verknüpft.

```
DEF Rot_Interpolator_ObjektId ORIENTATIONINTERPOLATOR {
KEY [ startzeit endzeit ]
KEYVALUE [ 0 0 0 0, a b c w ] }    // (a,b,c) Rotationsvektor, w Winkel
ROUTE Uhr.FRACTION_CHANGED TO RotInt_ObjektId.SET_FRACTION
ROUTE RotInt_ObjektId.VALUE_CHANGED TO ObjektName.SET_ROTATION
```

Kapitel 3

Auswertung

3.1 Implementierung

Die hier vorgestellten Konzepte wurden komplett unter Linux in C++ implementiert. OpenGL [6] diente als Grafikschnittstelle und die Benutzeroberfläche wurde mit QT realisiert. Eine Beschreibung von QT findet man im Buch der Entwickler [7]. Die erstellten Szenarios und Animationen kamen in den VR-Umgebungen Powerwall und CAVE zum Einsatz.

3.2 Anwendungsszenario

Das Anwendungsszenario besteht aus einem Produktionsbereich der Firma John Deere zur Herstellung von Teleskoparmen für Nutzfahrzeuge. Die Geometriedaten werden von John Deere zur Verfügung gestellt. Es soll eine Animation erstellt werden die drei einfache Arbeitsschritte simuliert. Dabei wird der Teleskoparm kontinuierlich an drei Arbeitsplätzen aus Einzelteilen zusammengesetzt. Gefordert ist lediglich eine grobe erste Annäherung. Kleinere Arbeitsschritte wie z.B. das Befestigen von Schrauben werden noch nicht berücksichtigt. Die Abbildungen 3.2 bis 3.4 zeigen die erstellte Animation.

3.3 Modularisierung

Die Modularisierung mittels VRML Extractor reduziert die Anzahl verschiedener Objekte von 597 auf 60. Aussortiert werden außerdem einige Hundert 'Dummy Objekte', das sind Objekte die keine Dreiecke enthalten. Zudem sind einige vollkommen identische Objekte mehrfach an derselben Position abgespeichert. Diese Objekte werden ebenfalls entfernt.

	vorher	nachher	Reduzierungsfaktor
Anzahl Objekte	597 Objekte	60 Objekte	1 : 9,95
Datengröße	36,9 MByte	7,9 MByte	1 : 4,67

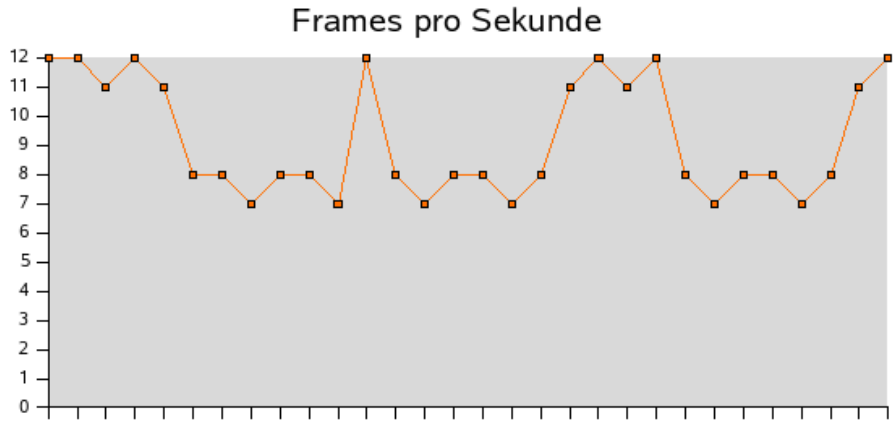

Abbildung 3.1: Performance

Die so modularisierten Daten schrumpfen von rund 36,9 Mbyte auf eine Größe von rund 7,9 Mbyte, inkl. der erzeugten KVP Masterdatei. Dabei gehen keine relevanten Daten verloren und die Daten sind deutlich besser strukturiert durch die Masterdatei. Der erzielte Reduzierungsfaktor liegt bei ca. 1 : 10 bzgl. der Objekte und bei ca. 1 : 4,67 bzgl. der Datenmenge. Siehe auch Abb. 2.3.

3.4 Zerlegung in Einzelteile

Da für den Zusammenbau nicht alle Einzelteile als eigene Objekte verfügbar sind, werden einige aus größeren Objekten ausgeschnitten. Es handelt sich dabei in unserem Beispiel um Einzelteile des Produktionsgegenstands, also des Teleskoparms. Siehe auch Abb. 2.4 und 2.5.

3.5 Animation

Die erstellte Simulation des Produktionsprozesses besteht aus 19 Animationen, bewegt werden 7 Objekte, die Animationsdauer beträgt 27 Sekunden. Die Hülle des Lader liegt auf einem Rollwagen, der am ersten Arbeitsplatz steht. Daran wird zuerst der ausfahrbare Arm befestigt. Dann fährt der Rollwagen samt Lader in den zweiten und dritten Arbeitsplatz. Dort wird dann in zwei Schritten das Werkzeug an den Arm montiert. Siehe Abb. 3.2 bis 3.4. Auf einem Athlon XP Mobile 2600 mit ATi Radeon 9600 Grafikchip und 1024 MB RAM lag die Framerate der Animation im Schnitt bei 9,5 Frames pro Sekunde (Abb. 3.1). Dieses Ergebnis ist auf die hohe Polygonzahl zurückzuführen. Der visualisierte Datensatz besteht insgesamt aus 116779 Polygonen. Mit Verfahren zur Reduzierung der Polygonzahl könnte man die Animation flüssiger gestalten.

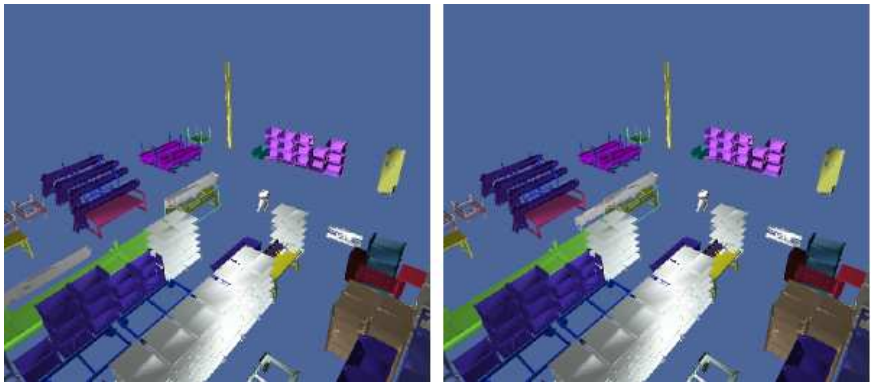

Abbildung 3.2: Hier sieht man wie der Teleskoparm zusammengesetzt wird

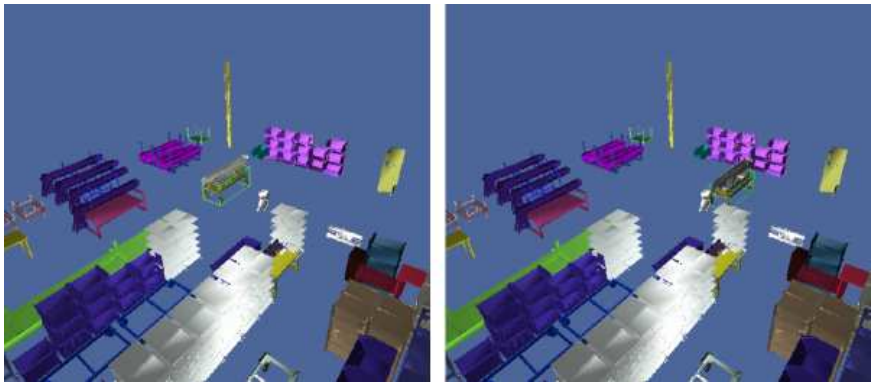

Abbildung 3.3: Im 3. u. 4. Frame kommt die Halterung für den Lader dazu

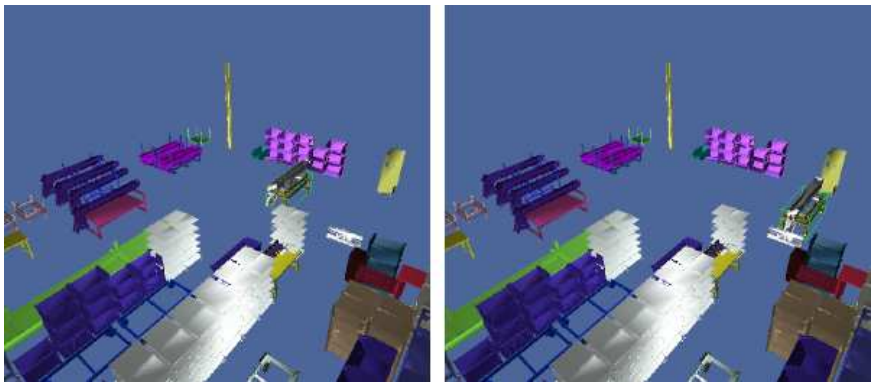

Abbildung 3.4: Die 2 letzten Frames zeigen wie die Ladeplattform montiert wird

Kapitel 4

Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Tool entwickelt das große VRML Datensätze von Produktionsprozessen modularisiert und organisiert, sowie ein weiteres Tool um Einzelteile aus Produktionsgegenständen auszuschneiden. Außerdem wurde ein Animationseditor vorgestellt der es erlaubt den Ablauf von Produktionsprozessen zu simulieren und zu verbessern. Diese Animationen kann man als VRML exportieren für eine interaktive Darstellung in einer immersiven VR-Umgebung (z.B. CAVE).

Aufbauend auf diesen Tools können Erweiterungen realisiert werden. Nützlich wäre sicherlich die Möglichkeit die Animationen nicht nur als Kombination von Translationen und Rotationen angeben zu können, sondern einfach dadurch daß man Wegpunkte angibt und diese zu einer Strecke interpoliert werden. Daraus müssten dann die äquivalenten elementaren Translationen und Rotationen abgeleitet werden. Ein weiterer Punkt wäre die Verfeinerung der Produktionsschritte mit mehr Details bis hin zur kleinsten Schraube, sowie eine Zoomfähigkeit mit detaillierter Darstellung von einzelnen Objekten. Hierbei wäre die Leistungsfähigkeit der gegebenen Hardware zu berücksichtigen (Level-of-Detail-Verfahren). Um die Übersichtlichkeit weiter zu gewährleisten wäre es sinnvoll das KVP Format als Baumstruktur aufzubauen. Als Wurzel das zu produzierende Objekt, als Knoten Zwischenprodukte und als Blätter die elementaren Einzelteile. Weiterhin wäre eine Integration von Simulationsmodulen interessant um bestimmte Parameter wie Temperaturverteilung oder Geräuschentwicklung zu visualisieren. Auch die Bedienbarkeit kann man weiter verbessern, z.B. durch Bewegen der Kamera relativ zu lokalen Objektkoordinatensystemen, oder das Einführen eines 'camera' Abschnitts im KVP Format um die Kamera schon beim Laden der Szene an die gewünschte Stelle zu bewegen.

Literaturverzeichnis

[1] Witt, J., Witt, T.: *Der kontinuierliche Verbesserungsprozess (KVP): Konzept - System - Maßnahmen.* Heidelberg: Sauer-Verlag 2001

[2] Wildemann, H.: *Kontinuierliche Verbesserung - Leitfaden zur kontinuierlichen Innovation und Verbesserung.* München: TCW Transfer-Centrum Verlag 1994

[3] Hofmann, J.: Raumwahrnehmungen in virtuellen Umgebungen - Der Einfluss des Präsenzempfindens in Virtual Reality-Anwendungen für den industriellen Einsatz. Wiesbaden: Deutscher Universitäts-Verlag 2002

[4] Burdea, G.; Coiffet, P.: Virtual Reality Technology, Second Edition. Hoboken (New Jersey): John Wiley & Sons, Inc. 2003

[5] The VRML Consortium Incorporated: *International Standard ISO/IEC 14772-1:1997.*

[6] Shreiner, D.: *The OpenGL Reference Manual: The Official Reference Document to OpenGL.* Addison-Wesley.

[7] Blanchette, J., Summerfield, M.: *C++ GUI Programming with Qt 3.* Bruce Peren's Open Source.