IAM-FCG-9601

Computer-Animation für BOOGA

Informatikprojekt
vorgelegt von
Thierry Matthey

angefertigt am
Institut für Informatik und angewandte Mathematik
Universität Bern

Betreuer:

Christoph Streit

Beginn der Arbeit: November 1995
Abgabe der Arbeit: Mai 1996

Zusammenfassung

Unter Computer-Animation versteht man eine Sequenz von bewegten Bildern, die mit Hilfe des Computers erzeugt wurden. Ziel einer Animation ist es eine Bewegung darzustellen. Eine Bewegung entsteht durch Manipulationen der Objekte, der Kamera und der Lichtquellen einer Szene bezüglich der Zeit. Eine Animation besteht einerseits aus einer Modellierung der Manipulationen und anderseits aus Methoden der Animationskontrolle.

Im Rahmen dieser Arbeit wurde auf BOOGA basierend Animationsklassen entwickelt, die eine explizite Animationskontrolle zulassen. Die Manipulationen können mit vielfältigen Funktionen algorithmisch modelliert werden. Für die Animationsbeschreibung wurde ein streng objektorientierter Lösungsansatz gewählt, d.h. es gibt keine Trennung der Szene und der Manipulationen.

Das vorliegende Grundgerüst ist mit den gängigsten 3D-Transformationen relativ einfach gehalten, sollte aber leicht erweiterbar sein. Als Erweiterung ist auch eine darüberliegende Schicht denkbar. Zusätzlich wurde eine schon vorhandene Applikation für die Visualisation von Animationen und das Generieren von MPEG-Dateien umgeschrieben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Ziele

Für das Projekt als ganzes haben mein Betreuer und ich folgende Ziele gesteckt:

Lernziele

- Die Programmiersprache C++ und ihre OO-Elemente erarbeiten.
- Der Umgang mit Programmiertools wie SNiFF+.
- Selbständiges Arbeiten, Eigenentwicklung.
- Sinnvolles Dokumentieren.

Implementation

- Das Projekt Computer-Animation für BOOGA soll es ermöglichen, mit einfachen Mitteln Animationen zu erstellen.
- Mit dem Gelingen des Projektes soll BOOGA vorangetrieben werden.
- Code, der funktioniert.

Als Nicht-OO-Kenner war der Zugang nicht gerade leicht, weil BOOGA auf C++ und der OO-Technik basiert. Anfangs verstand ich nur wenig von BOOGA, weil für mich die ganze Vererbung und der Klassenaufbau neu waren. Da das Framework schon sehr viele Klassen enthält und darin mehrere Applikationen implementiert sind, konnte ich mich an typischen Beispielen wie `Sphere3D.C` bzw. `flythrough.C` orientieren und abgucken. Trotz allen Beispielen wurde ich erst beim Design und der Implementation der Animationsklassen im OO-Bereich richtig gefordert und musste für beispielhafte Anfängerfehler viel Zeit investieren. Dank der OO-Technik konnte ich den Programmier- und Codeaufwand stark reduzieren. Da mein Projekt auf BOOGA aufbaut und schon eine Applikation für die Darstellung von Szenen wie `flythrough` implementiert war, konnte ich mich ganz auf die Entwicklung von Animationskonstrukten konzentrieren, ohne dass ich mich um die Ein- und Ausgabe kümmern musste. Somit war es mir auch möglich, das Implementierte gleich zu testen, wobei das Testen nicht immer einfach war. Gewisse Fehler und Ungenauigkeiten waren nicht immer von blossem Auge erkennbar. Das Tool SNiFF+ half mir, eine Übersicht über BOOGA zu gewinnen und war mir beim Editieren, Kompilieren, Debugen und Browsen sehr behilflich. Sehr gefallen haben mir die zusätzlichen Tools von SNiFF+, die z.B. ein gleichzeitiges Vergleichen und Editieren von zwei Programmversionen zulassen. Auch wenn SNiFF+ gewisse Macken hat und zum Teil etwas träge ist, möchte ich es nicht mehr missen. Die relativ spärliche Dokumentation von BOOGA , fehlendes Hintergrundswissen und die Komplexität des Projektes machten den Start für die Entwicklung von eigenen Ideen etwas harzig. Trotz diesen Startschwierigkeiten konnte ich einige Ansätze entwickeln und evaluieren, um zu einem befriedigenden Ergebnis zu kommen. Durch das relativ offene Projekt wurde das fortlaufende Dokumentieren zum Sammeln von Ideen, Skizzen, Problemen, Programmversionen, etc. und die vorliegende

Dokumentation entstand, wie so oft, erst am Schluss.

Die festgelegten Ziele habe ich im grossen und ganzen erreicht, auch wenn noch Verbesserungen möglich wären. Leider konnte ich aus Zeit- und Komplexitätsgründen keine Transformationen der Oberflächeneigenschaften der Objekte implementierten ("wäre schön, muss aber nicht unbedingt sein"). Dieses Projekt betrachte ich als ersten Schritt in Richtung Computer-Animation für BOOGA , und es kann für andere Arbeiten als unterste Schicht oder bei Erweiterung als Startpaket dienen. Es sei schon hier vorweggenommen, dass eine gute Animation auch von guten Ideen und Kreativität lebt.

Das Projekt wurde zeitlich in fünf Zwischenziele aufgeteilt:

1. Einbau einer Timerfunktionalität in die Applikation `flythrough`, die einfache Kamerabewegungen ausführt.
2. Automatische Veränderung der Form, Farbe, Texture, etc. eines Objektes.
3. Funktionalität für das Generieren von MPEG-Dateien.
4. Entwicklung von Animationsklassen mit einem einfachen `tick()`-Protokoll, das ein einfaches Inkrementieren einer Transformation zulässt.
5. Die umgebaute `flythrough` Applikation sammelt alle Animationsobjekte und führt auf Grund eines Ereignisses die `tick()`-Methode aus.

Die Zwischenziele dienten mir als Kontrolle für das Gelingen des Projektes. Damit war es möglich, mir über das Erreichen eines Zwischenzieles Rechenschaft abzulegen und nötige Kurskorrekturen und Anpassungen vorzunehmen. Zusätzlich hatte ich mit meinem Betreuer Besprechungen, die mich zwangen, über den Stand der Dinge zu reflektieren.

Die ersten drei der fünf Phasen konnte ich nach meinem selbsterstellten Zeitplan durchführen. Die letzten zwei Phasen waren arbeitsintensiver, als ich geplant hatte. Mit den OO-Unkenntnissen, der ganzen Animationsproblematik und der Offenheit des Projektes wurden Design und Implementation zu einem iterativen Prozess. So war es aber auch möglich, neue Ideen einfliessen zu lassen, zu implentieren oder zu verbessern. Dieser iterative Prozess liess sinnvollerweise nur noch ein Sammeln von Ansätzen und ihren Qualitäten, Ideen, etc. zu. Leider ist es auch mir schwergefallen, zwischen "unbedingt nötig" und "wäre schön" eine klare Grenze zu ziehen. Trotz allen Verzögerungen konnte ich die Arbeit dank zwei Monaten Pufferzeit rechtzeitig abgeben.

1.2 Aufbau der Arbeit

Die Einleitung erläutert die Ausgangslage, die Vorgehensweise und die Hilfsmittel des Projektes. Das zweite Kapitel gibt eine Einführung in die Computer-Animation und erklärt verschiedene Ansätze. Im dritten Kapitel wird auf die Implementierung eines Ansatzes und die Einbettung in BOOGA eingegangen. Das vierte Kapitel erklärt anhand eines Beispieles die nötigen Schritte von der Idee bis zum fertigen MPEG-Film. Das letzte Kapitel gibt einen kurzen Überblick über Problemfälle und mögliche Erweiterungen, die bei der Bearbeitung des Projektes auftraten. Das Literaturverzeichnis enthält die relevanten

Bücher und Skripte des Projektes. Der Anhang enthält Bilder von Animationen mit den entsprechenden BSDL-Dateien.

1.3 BOOGA (Berne's Object-Oriented Graphics Architecture)

Das Framework BOOGA ist objektorientiert und deckt einen grossen Teil der Computergrafik ab. Es soll die Entwicklung von Applikationen vereinfachen, indem es einerseits standardisierte, wiederverwendbare Softwarekomponenten enthält und anderseits das Entwickeln und das Implementieren neuer Komponenten unterstützt. BOOGA dient auch als Forschungsplattform zur Entwicklung neuer Techniken und Algorithmen. Durch die vielfältigen Anforderungen wird eine flexible Plattform gefordert, die einfach erweiterbar ist. Damit wird ein hoher Wiederverwendungsgrad erreicht. Es sollte auch möglich sein, Resultate der Entwickler in Folgearbeiten ohne Änderung wiederverwenden zu können. Das Gebiet von BOOGA umfasst Teilbereiche der 2D und 3D Computergrafik. Es werden die Objektmodellierung für 2D und 3D, die Bildgenerierung und die Bildanalyse unterstützt. Das Design der 2D und 3D Grafik ist so weit als möglich vereinheitlicht. Die Architektur

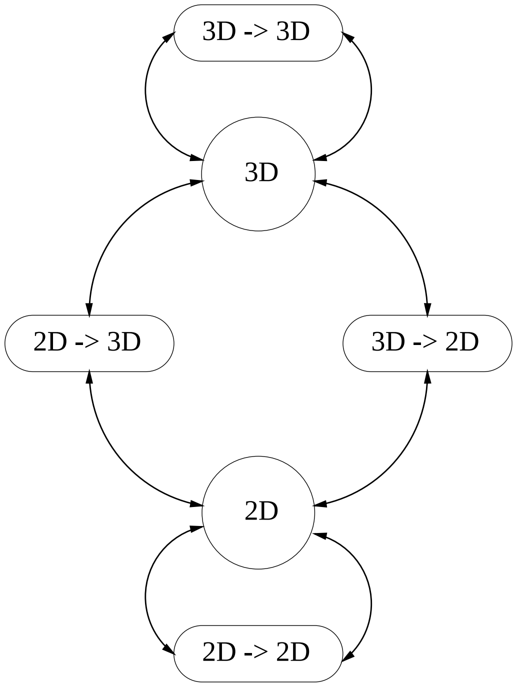


Abbildung 1: Datenflussmodell von BOOGA.

basiert auf einem einfachen Datenflussmodell (Abb. 1). Die Daten sind 2D bzw. 3D Szenen, in BOOGA Welten genannt. Die möglichen Operationen und Übergänge der Welten werden Komponenten genannt. Es wird von vier Komponententypen ausgegangen:

- Arbeiten mit einer 2D Welt.
- Erzeugen einer 3D Welt anhand einer 2D Welt.
- Arbeiten mit einer 3D Welt.

- Erzeugen einer 2D Welt anhand einer 3D Welt.

Die Einfachheit des Konzeptes lässt eine vielfältige Abdeckung der Anwendungsgebiete zu. Die Erweiterbarkeit und die komponentenorientierte Softwareentwicklung wurde durch ein objektorientiertes Design sowie durch den Gebrauch passender Design Patterns erreicht.

1.4 Hilfsmittel

Als Programmiertool verwendete ich, wie alle anderen, die mit BOOGA arbeiten, SNiFF+ 2.1, das mir das Kompilieren, Editieren und Debugen sehr stark erleichterte. SNiFF+ enthält vielfältige Tools zur Bearbeitung von Grossprojekten. Zum einem wird gleichzeitiges Entwickeln mit eigenem Workspace und eigenem Entwicklerstatus unterstüzt, zum andern enthält SNiFF+ einen Hierachie- und Klassenbrowser, die den Umgang mit den Klassen erleichteren, sowie Tools, die das Updaten oder das Vergleichen zweier Versionen übernehmen. Unter SNiFF+ wurde der relativ strenge C++-Compiler GNU-Compiler 2.7.2 benutzt. Unerlaubte und unkorrekte Speicherzugriffe wurden mit Hilfe von Purify Version 3.2 ausfindig gemacht. Purify ist auch bei der Suche nach nicht freigegebenem Speicher behilflich.

Als Basis und Plattform für die Entwicklung wurde BOOGA verwendet, das viele wichtige Ausgabemöglichkeiten des Renderings abdeckt und das eine Eingabesprache (BSDL - Booga Scene Description Language) für die Modellierung von Szenen mit Objekten enthält. Dank BOOGA konnte ich von schon implementierten Applikationen und Klassen viel profitieren. Sie dienten mir als Beispiel für die Entwicklung von Applikationen und Klassen.

Für die Erstellung von MPEG-Dateien wurde die MPEGelib 0.1 von Alex Knowles (`http://www.tardis.ed.ac.uk/~ark/mpegelib/`) benutzt, die eine einfaches Generieren von MPEG-Filmen ermöglicht. Diese Library basiert auf dem `mpeg_encoder` von Berkley und übernimmt die ganze mühselige Steuerung des `mpeg_encoder`. Es hat sich aber gezeigt, dass diese Library etwas eingeschränkt ist und die Qualität des `mpeg_encoder` nicht immer erreicht.

Die Dokumentation editierte ich mit Word und portierte sie dann nach LaTeX , was mir eine sichere und einfache Verwaltung der Kapitel, der Inhaltsverzeichnisse etc. bot. Für die eingebundenen Figuren benutzte ich XFIG, das importierbare Dateien für LaTeX erzeugt. Damit war es mir mit nicht all zu grossem Aufwand möglich, ein professionelles Layout zu erstellen.

1.5 Acknowledgements

Als erstes möchte ich meinem Betreuer Christoph Streit danken. Er hat sich, wann immer möglich, die Zeit genommen, mir jede Frage ausführlich und nach bestem Wissen zu beantworten. Auch lehrte er mir die Tricks, die einen Informatiker am Leben erhalten.

Als nächstes gebührt mein Dank allen FCG Mitarbeitern, die mich mit Tips & Tricks

und guten Ideen unterstützt haben.

Schliesslich seien auch all jene erwähnt, die sich für mein Projekt interessierten und dadurch manche Anregungen an mich weitergeben konnten.

Speziell möchte ich Axel Knowles aus England für die promte Lieferung der MPEGe-lib 0.1 und die vielen anregenden Briefwechsel danken.

Last but not least möchte ich noch meinem norwegischen Freundeskreis und speziell meiner Freundin danken, die es mir ermöglichten, von Norwegen aus mein Projekt voranzutreiben.

2 Einführung in die Computer-Animation

Die ersten bewegten Bilder entstanden schon in der Mitte des 19. Jahrhunderts. Doch die Entwicklung der Animation begann erst Anfangs 20. Jahrhundert, nach der Einführung des Films. 1915 führte der Amerikaner Earl Hurd die Technik der *cel animation* ein, welche ihren Namen von dem transparenten Zelluloidpapier erhalten hat. Die erste kommerzielle Animation mit synchronisiertem Ton entstand 1928 unter Walt Disney.

Heute versteht man unter Animation eine Sequenz von bewegten Bildern, die mit Hilfe des Computers erzeugt werden. Die Bewegung von Bildern entsteht durch Veränderungen von Standort (*motion*), Form (*shape*), Farbe, Transparenz, Struktur, Textur eines Objektes, des Kamerastandortes oder der Lichtquellen. Die Einzelbilder, auch Frames genannt, bilden zusammen eine Bildsequenz. Bei der Veränderung von Bild zu Bild spricht man von der Dynamik (*motion dynamic*) der Animation, die oft als maximale Verschiebung eines Pixels im Bildraum definiert wird. Die Erstellung von Computer-Animationen geschieht mit Hilfe von Computer-Animationssystemen.

2.1 Klassifizierung der Computer-Animationssysteme

Es gibt eine Vielzahl von Klassifizierungsmöglichkeiten von Computer-Animationssystemen. Die folgende Liste gibt einen möglichen Überblick [MTT90]:

Stufe 1	:	Primitives Malprogramm, das nur einfaches Modifizieren eines Bildes erlaubt.
Stufe 2	:	Berechnung von Zwischenbildern (*in-betweens*) und Verschiebung von Objekten entlang eines Pfades.
Stufe 3	:	Transformation der Objekte und virtuelle Kamera.
Stufe 4	:	Animationsobjekte kommunizieren untereinander und verändern sich selbst.
Stufe 5	:	Expertensystem, lehrnfähiges System.

Die Stufen 3 bis 5 verwenden im Normalfall einen abstrakten 3D-Objektraum mit virtueller Kamera. Dieser 3D-Objektraum ist dabei fundamental, deshalb wird ab jetzt immer ein System der Stufe 3 angenommen, wenn nichts Spezielles erwähnt wird.

Mit Systemen der Stufe 1 und 2, normalerweise ohne Objektraum, entsteht eine Bildsequenz durch Transformationen im Ausgangsbild (Abb. 2). Eine Transformation wird auf der Pixelebene ausgeführt, indem die einzelnen Pixels ihren Ort oder ihre Farbe ändern. Eine Bewegung kann beispielsweise schon mit einem Malprogramm mit den Funktionen `Cut` und `Paste` erzielt werden, indem man die gewünschte Figur im Bild verschiebt und die einzelnen Bilder speichert. Ausgereifte Grafikprogramme kennen heute elegante Effekte, wie das Aufrollen eines Bildes, Wirbelbewegungen (Abb. 3) oder können Übergänge (*metamorphosis*) zwischen zwei Objekten berechnen. Applikationen der zweiten Stufe unterstützen die Berechnung von Zwischenbildern und automatisieren Verschiebungen von Objekten entlang eines vorgegebenen Pfades. Damit können einfachere Animationen erstellt werden, wobei der Zeichner selber für eine korrekte Perspektive sorgen muss. Heute unterstützen Malprogramme auch ein pseudo-perspektivisches Zeichen ($2\frac{1}{2}$D) und das Expandieren von 2D-Objekten in die dritte Dimension. Zusätzlich kann sich auch das Ausgangsbild im Laufe der Zeit selbst ändern. Solche Animationen oder Effekte trifft man

Abbildung 2: Ausgangsbild.

Abbildung 3: Anwendung der Twirlfunktion.

heute oft in Videoclips oder in TV-Reklamen an. Dieser Ansatz, ohne Objektraum, lässt interessante Möglichkeiten zu, ist aber bei lokalen Veränderungen mit globalen Auswirkungen schlecht anwendbar. Beispielsweise verändert die Verschiebung der Sonne in einem Bild alle Schattenflächen. Bei komplexen Transformationen oder Übergängen müssen viele Stützpunkte eingegeben werden, damit die Transformation korrekt ausgeführt werden kann, ohne benachbarte Regionen zu verändern. Der Grund für diese Schwerfälligkeit ist die Repräsentation des Bildes als Rastergrafik, die keine abstrakte Darstellung von Objekten kennt.

Für Systeme ab Stufe 3 wird von einem Objektraum ausgegangen, der dann in einen Bildraum projiziert wird. Für Computer-Animationen ist dieser Raum meistens R^3. Die Szene wird in diesem abstrakten Raum mit Objekten modelliert. Die Grundobjekte, die oft

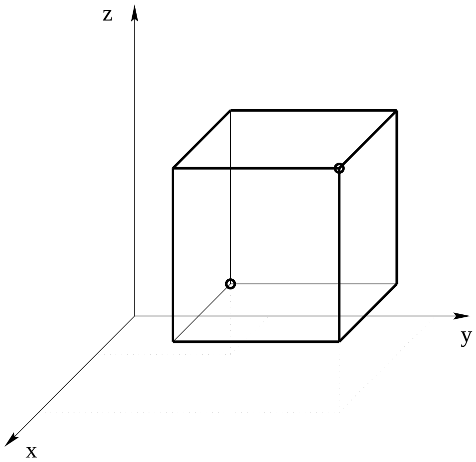


Abbildung 4: Definition eines Quaders im Objektraum R^3.

Primitive (*primitiv*) genannt werden, können zu komplexeren Objekten zusammengesetzt werden. Die Definition eines Grundobjektes ist vektororientiert, d.h. ein Quader (Abb. 4) wird beispielsweise durch Angabe von zwei gegenüberliegenden Eckpunkten definiert. Ein Primitiv kann noch Attribute wie Farbe, Textur oder Transparenz enthalten. Diese abstrakte Darstellung einer Szene ist besonders für das Manipulieren der Objekte geeignet. Die Abbildung in den Bildraum (Betrachtungstransformation) wird durch die Darstellungsart, die Projektion, den Kamerastandort und die Blickrichtung (*look direction*) definiert. Parallelprojektion und Perspektive sind die zwei wichtigsten Projektionsarten. Mit der Darstellungsart bestimmt man das Aussehen und die Darstellung der Objekte selbst. Die Darstellungsart kann von Wireframe- bis zu aufwendigen Raytracing-Darstellungen (Abb. 6 bzw. 5) variieren. Kamerastandort und Blickrichtung bestimmen das Blickfeld des Betrachters. Eine Animation entsteht, indem die modellierte Szene fortlaufend im Objektraum verändert und neu berechnet wird.

2.2 Computer-Animation

Die Erstellung von Computer-Animationen, die auf einem Objektraum aufbauen, beinhaltet folgende drei Hauptaktivitäten:

1. Modellierung der Objekte im Objektraum.
2. Spezifikation der Transformationen.
3. Darstellung der animierten Szene.

Die Modellierung der Szene und die Spezifikation der Transformationen kann über eine grafische Oberfläche geschehen, die ein einfaches und schnelles Arbeiten erlaubt. Dazu sind keine Programmierkenntnisse nötig, weshalb grafische Oberflächen für reine Anwender geeignet sind. Computer-Animationssysteme mit einer Animationssprache wirken etwas umständlich, weil die Animation als Skript vorliegen muss und die Modellierung der Szene der Programmierung gleichkommt. Diese Systeme sind aber oft viel flexibler und ausgiebiger. Die Sprache mit ihren Animationskonstrukten erlaubt eine genau Spezifikation der Transformationen der Objekte. In Szenen von Computer-Animationen können die Eigenschaften der Kamera und der Lichtquellen editiert bzw. programmiert werden.

Eine Bildsequenz kann einerseits in Echtzeit (*realtime*) und anderseits Bild um Bild (*frame-by-frame*) entstehen. Unter Echtzeit-Animation versteht man Animationen, die in Echtzeit berechnet und angezeigt werden. Echtzeit wird oft für das Visualisieren von Bewegungsabläufen oder von interaktiven Prozessen benutzt. Man verzichtet hier der Geschwindigkeit zu liebe bewusst auf detaillierte Bilder. Wichtig ist die aktuelle und rechtzeitige Anzeige einer Situation oder Szene. Für die Illusion eines flüssigen Verlaufs reichen dem menschlichen Auge schon 15 Bilder pro Sekunde. In heutigen Anwendungen sind deshalb 15 Bildern pro Sekunde die Mindestanforderung. In solchen Applikationen wird das Bild fortlaufend anhand von irgendwelchen Eingabedaten oder Interaktionen des Benutzers neu berechnet. Die heutigen Computer mit Hardwarebeschleuniger erreichen schon ansprechende Resultate, aber Echtzeit-Animationen mit Raytracing-Qualität sind noch ferne Zukunftsmusik. Das Erstellen Bild um Bild (Bild-um-Bild-Animation) wird für Anwendungen eingesetzt, die eine hohe Bildauflösung und eine detaillierte Darstellung erfordern

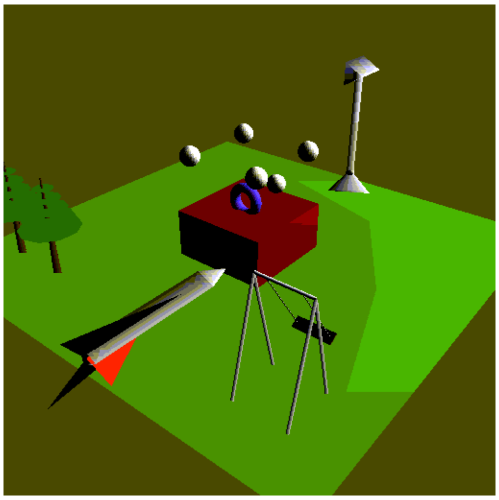


Abbildung 5: Szene mit Solid Gouraud.

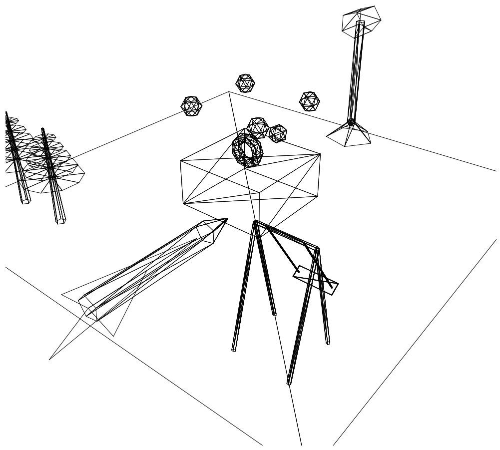


Abbildung 6: Szene mit Wireframe.

und keinen Anspruch auf die Berechnung der Bilder in Echtzeit haben. Diese Methode ist für die Filmherstellung und auch für Anwendungen, die ohne Interaktion auskommen, bestens geeignet. Bei der Berechnung der Bilder werden rechenintensive Effekte wie Schatten, Transparenz oder Reflexion berücksichtigt, mit dem Ziel, realitätsnahe Bilder zu erhalten. Die ganze Bildsequenz wird einmal Bild um Bild berechnet, gespeichert oder sogar als Film belichtet. Danach kann die Sequenz in Echtzeit nach belieben abgespielt (*real-time playback*) werden.

Die Illusion einer Bewegung kann noch auf andere Art erzielt werden. Viele Methoden entstanden mit dem Aufkommen der Zeichentrickfilme, deren Herstellung zu jener Zeit sehr mühselig war, weil jedes einzelne Bild einer Sequenz von Hand gezeichnet und eingefärbt wurde. Man begann, zyklische Abläufe einmal zu zeichnen und brachte den Zyklus so oft unter die Kamera, wie er schliesslich im Film erscheinen sollte. So konnte aus einer kleinen Menge von Zeichnungen ein Film (*cycle animation*) gestaltet werden. Ein typischer Vertreter von vereinfachten Animationen, der sich dieser Technik bedient, sind die Fred Feuerstein-Trickflime (*reduced animation cartoons*). Eine weitere Möglichkeit in der Computer-Animation ist das zyklische Vertauschen von Farben im Bild (*color cycling*). Damit kann zum Beispiel fliessendes Wasser imitiert oder Punkte zum Laufen gebracht werden (Abb. 7), oder man fasst alle Bitplanes der Bilder zu einem Bild zusammen und schaltet die entsprechenden Bitplanes mit ihren Farben ein und aus. Als letzte Möglichkeit sei noch das Verwenden eines übergrossen Bildes erwähnt. Dabei wird nur ein kleiner Ausschnitt angezeigt, der über das ganze Bild wandert und so einen Animations-Effekt bewirkt. Diese Methoden werden oft bei zyklischen Bewegungen verwendet, um sich wiederholende Bilder zu ersparen. Ein Bewegung entsteht dann durch mehrfaches Abspielen eines ganzen Zyklus.

Eine Mischform aus Objektraum und Manipulation von Rastergrafik wird heute bei Video- und Computerspielen verwendet. Diese Spiele stellen hohe Anforderungen an die Entwickler, da realitätsnahe Bilder in kürzester Zeit generiert werden müssen. Die Ressour-

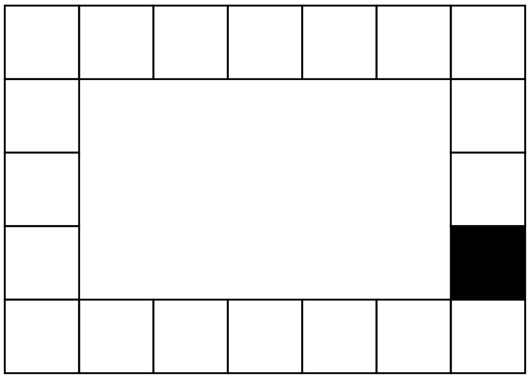


Abbildung 7: Color Cycling.

cen müssen aber nicht nur für die Bilddarstellung reichen, sondern auch für das Abarbeiten der Interaktion durch den Spieler. Um die rechtzeitige Bilddarstellung zu erreichen, wird dann oft eine Mischform mit gewissen Einschränkungen benutzt. Der Objektraum und die Objekte werden vereinfacht, indem ein diskreter Objektraum verwendet wird und nur einfache Objekte zugelassen werden. Eine Szene wird dann aus quadratischen Standardelementen aufgebaut und nachträglich mit realitätsnahen Texturen eingefärbt. Oft wird eine Zentralperspektive mit eingeschränktem Blickfeld gewählt.

2.3 Qualität einer Animation

Die Qualität einer Animation hängt von den Ideen und der Kreativität und von technischen Faktoren ab. Die technische Qualität wird hauptsächlich von den vier Faktoren Bildauflösung (*resolution*), Darstellung, Dynamik und Bildwiederholungsfrequenz (*framerate*) bestimmt.

Die Bildauflösung wird durch die Höhe, die Breite und die Farbenvielfalt des Bildes bestimmt. Der Rechenaufwand steigt etwa linear zur Höhe und Breite des Bildes an. Für den Speicherungsaufwand kommt noch die Farbenvielfalt dazu, ohne auf eventuelle Komprimierungsverfahren Rücksicht zu nehmen. Das Komprimieren kann den Speicheraufwand stark reduzieren, allerdings nur auf Kosten des Rechenaufwandes. Diese Verfahren sind für die Speicherung von Filmsequenzen und Anwendungen, die keinen Anspruch auf Echtzeit haben, bestens geeignet.

Die Darstellung der Szene wird durch die Betrachtungsfunktion und die Darstellung der Objekte gegeben. Dabei kann man zwischen einfachen Darstellungen wie Wireframe und anspruchsvolleren Raytracingmethoden wählen. Die Darstellungsart lässt die grösste Variation bezüglich Rechenaufwand und Qualität zu. Eine einfache Darstellung wie Wireframe eignet sich deshalb sehr gut für Echtzeitanwendungen mit grosser Dynamik und hoher Bildwiederholungsfrequenz. Eine anspruchsvollere Methode wie Raytracing ist für das Generieren von Filmen geeignet, verlangt aber einen ungemein höheren Rechenaufwand.

Die Dynamik gibt die maximale Änderung von Bild zu Bild wieder. Oft wird die maximale Verschiebung eines Pixels zwischen zwei Bildern gemessen. Bei all zu grosser Dynamik mit all zu kleiner Bildwiederholungsfrequenz beginnt die Animation zu rucken. Deshalb sollte darauf geachtet werden, dass die Dynamik bei tiefen Frequenzen nicht all zu gross wird.

Mit der Bildwiederholungsfrequenz wird die Anzahl der wiedergegebenen Bilder pro Sekunde angegeben. In der Filmwelt arbeitet man typischerweise mit 24 Bildern pro Sekunde, wobei für Animationen schon 15 Bilder pro Sekunde ausreichen. Der Arbeits- und Speicheraufwand steigt linear zur Frequenz an. Es empfiehlt sich bei grosser Dynamik die Frequenz hoch zu halten und falls nötig eine einfachere Darstellungsart zu wählen.

2.4 Wiedergabe der Animation

Die Wiedergabe ist ein wichtiges Gebiet der Computer-Animation und besteht nicht nur aus der Anzeige von Einzelbildern. Die Anzeige von Einzelbildern bedingt eine Interpretation der wiederzugebenden Animation. Nebst der Bildausgabe müssen die Einzelbilder untereinander synchronisiert werden, damit ein korrekter Ablauf entsteht. Bei vertonten Animationen besteht die Synchronisation auch aus dem Abstimmen von Bild und Ton.

Echtzeitanwendungen bringen bei der Wiedergabe den grössten Interpretationsaufwand mit sich. Die Einzelbilder liegen nicht als Rastergrafik vor und müssen aus abstrakten Eingabedaten fortlaufend (Bild um Bild) berechnet werden. Die relativ aufwendige Berechnung der Einzelbilder, die durch die Bildwiederholungsfrequenz eingeschränkt wird, muss in nützlicher First geschehen, damit ein flüssiger Bewegungsablauf entsteht. Die minimale Bildwiederholungsfrequenz liegt bei etwa 15 Bildern pro Sekunden, damit bei all zu schnellen Veränderungen kein Rucken (*temporal aliasing*) auftritt. Bei Echtzeitanwendungen geht es mehr darum, die Bewegungsabläufe zu studieren, als Details von Objekten zu begutachten. Es hat sich auch gezeigt, dass das menschliche Auge auf wohlgestaltete Bewegungsabläufe besser anspricht als auf detaillierte Objekte. Deshalb sollte man bei schnellen Bewegungen oder grosser Dynamik die Darstellungsart entsprechend vereinfachen, um eine möglichst hohe Bildwiederholungsfrequenz zu erreichen. Eine zusätzliche Beschleunigung der Ausgabe kann man auch durch eine Beschränkung der Anzahl der darzustellenden Objekte oder durch eine Vereinfachung der Objekte erreichen. Vereinfachungen von Objekten erreicht man durch eine beschränkte Darstellung oder mit einer reduzierten Genauigkeit für entfernte oder kleine Objekte. Diese Philosophie wurde schon unter *BOOGA* in der Applikation `flythrough` implementiert.

Bei der Wiedergabe von Animationen, die nicht in Echtzeit berechnet werden, werden die Einzelbilder im allgemeinen aus einer oder mehreren Eingabedateien extrahiert. Hier wird grundsätzlich zwischen der Echtzeit-Wiedergabe (*real-time playback*) und der Wiedergabe von Animationen (*frame buffer animation*), die nicht als Bildfolge vorliegen, unterschieden. Bei der Echtzeit-Wiedergabe wird Bild um Bild angezeigt (*frame-by-frame*). Die Bilder werden dabei aus einer Datei extrahiert oder liegen als Einzelbilder vor und müssen noch in den Framebuffer geladen werden. Im Normalfall wird die Animation mit Komprimierungsverfahren in eine Datei gepackt, um den Speicheraufwand zu reduzieren. Komprimierungsverfahren haben aber den Nachteil, dass bei der Erstellung und bei der

Wiedergabe ein zusätzlicher Rechenaufwand entsteht. Dieser Nachteil kann sich dann speziell bei der Wiedergabe bemerkbar machen, wenn die Animation nicht mehr in Echtzeit wiedergegeben werden kann. Deshalb werden heute Dekomprimierungsverfahren als Hardwarekomponente realisiert, um solche Verzögerungen zu vermeiden. Der wichtigste Punkt, der für die Komprimierung spricht, ist der grosse Datenstrom zwischen Speicherplatte und Hauptspeicher, der bei anspruchsvolleren Animationen ($600 \cdot 400 \cdot 256 \cdot 15 \approx 880$ [MB/s]) ohne Komprimierung das System hoffnungslos überfordert. Die meisten Verfahren arbeiten mit Bildqualitätsverlust, um einen höheren Komprimierungsfaktor zu erzielen. Beim Abspielen der komprimierten Sequenz sind die Fehler von blossem Auge nicht erkennbar und deshalb auch vernachlässigbar. Ein weit verbreitetes Verfahren ist MPEG, das mit Bildqualitätsverlust arbeitet und die Kohärenz der Bilder ausnutzt. Bei vertonten Animationen gestaltet sich die Synchronisation von Bild und Ton nicht einfach, da der Ton nicht ohne triviale Operationen zeitlich skaliert werden kann. Das Bild muss deshalb auf den Ton abgestimmt werden. Es gibt heute schon Methoden, die Ton und Bild in eine Datei packen. Bei der Wiedergabe muss man aber darauf achten, dass die Wiedergabefrequenz stimmt.

Eine andere Möglichkeit ist das Vortäuschen einer Bild-um-Bild-Animation mit der Technik von *cycle animation* (siehe Seite 9). Die Animation wird in den Hauptspeicher geladen und entsprechend der Methode abgespielt. In solchen Sequenzen gehen oft Anfangs- und Endbild ineinander über. So können endlose Animationen mit geringem Speicher- und Rechenaufwand wiedergeben werden. Beispielsweise kann in einem Computerspiel ein Hintergrund mit Wasserfall mit *color cycling* imitiert werden, ohne die Computerressourcen merklich zu belasten.

2.5 Beschreibung der Animation

Mit der Beschreibung der Animation wird die Eingabe und die Erstellung von Animationen bestimmt. Die Beschreibung selber wird aber durch die Wahl des Animationsansatzes bestimmt. Der Animationsansatz besteht aus der Modellierung der Transformationen und aus den Methoden der Animationskontrolle. Dieser Abschnitt erklärt das Zusammenwirken der Animation und der Szene.

Die Anforderungen an eine Beschreibung der Animation gehen von einfachen, effektiven und wenn möglich zusammensetzbaren Animationskonstrukten aus, die eine Animation klar und eindeutig definieren. Das Ziel einer Beschreibung ist es, mit einer kleinstmöglichen Informationsmenge die gewünschte Transformation beschreiben zu können. Eine Animation kann mit Hilfe eine grafischen Oberfläche interaktiv editiert oder mit einem Skript beschrieben werden. Beide Varianten haben ihre Vor- und Nachteile. Eine grafische Oberfläche ist zwar einfach zu bedienen, kann aber für das Erstellen von komplexen Animationen umständlich sein. Das Skript lässt fast alle Möglichkeiten offen, ist aber oft schwer zu verstehen. Programmierkenntnisse sind oft unabdingbar. Um jegliche Nachteile umgehen und alle Vorteile nutzen zu können, hat man beide Methoden zusammengefasst. Die grafischen Oberflächen sind mit den Funktionalitäten des Importierens und des Exportierens von Skripten erweitert worden.

Mit der Beschreibung der Animation muss aber auch das Zusammenspiel von Szene

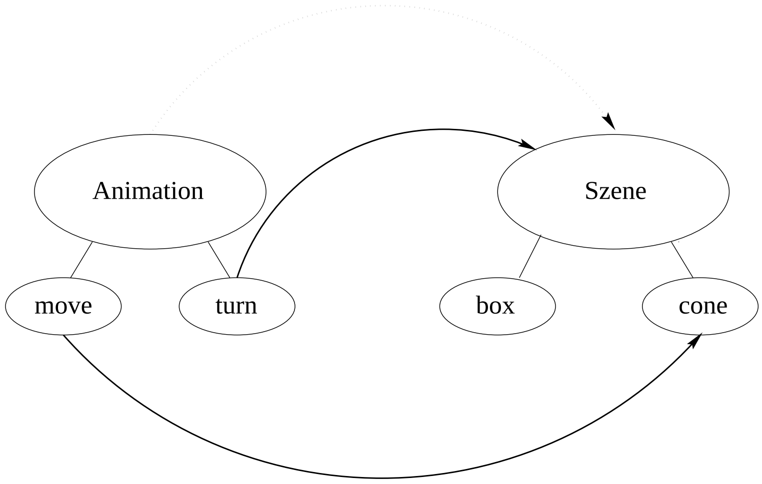


Abbildung 8: Trennung von Animation und Szene.

und Animation bestimmt werden. Das Zusammenspiel der Animation und der Szene wird stark durch die Wahl des Animationsansatzes bestimmt. Eine Möglichkeit ist die strikte Trennung von Szene und Animation (Abb. 8). Die Animation übernimmt dann die ganze Kontrolle über die Szene und verändert sie anhand der Eingabe. Damit die Applikation, die die Bildsequenzen erzeugt, weiss was es wie bewegen oder verändern soll, muss in der Eingabe auf die betreffenden Objekte der Szene referenziert werden. Das Referenzieren auf die Szene geschieht je nach gewähltem Ansatz unterschiedlich. Bei Key-Frame-Animationen beispielsweise wird auf ein Key-Frame referenziert. Dabei wird meistens zwischen Hintergrund und den zu animierenden Objekten unterschieden, um den Speicher- und Rechenaufwand zu reduzieren. Bei grafischen, interaktiven Oberflächen wird oft einem Objekt eine Transformation mit ihren Parametern zugeordnet. Eine andere Möglichkeit ist die Gleichbehandlung und das Zusammenfassen der Szene und der Animation (Abb. 10 Seite 18). Die Szene wird hierarchisch aufgebaut und enthält entsprechende Animationsobjekte, die dann auf darunterliegende Objekte Einfluss nehmen können. Der strikte hierarchische Aufbau der Szene erlaubt eine klare Abgrenzung des Wirkungsgebietes eines Animationsobjektes. Dieser Ansatz eignet sich für Animationstechniken, die von einer Szene ausgehen und alle Frames daraus erzeugen. Ein weiterer Ansatz ist die 4D-Modellierung, die keinen Unterschied zwischen Raum und Zeit macht. Alle Primitive (grafische Grundobjekte) werden als 4D-Objekte realisiert und die Animation wird durch die Verschiebung einer Hyperebene im 4D-Objektraum definiert.

2.6 Modellierung von Transformationen

Bei der Modellierung der Transformation beschäftigt man sich mit der Art, wie ein Objekt transformiert wird und mit der Dynamik der Transformation. Eine Transformation ist entweder eine Bewegung, die Ort und Ausrichtung eines Objektes verändert, oder eine Deformation, die ein Objekt in seiner Form verändert. Die Modellierung kennt nach [MTT90] zwei Hauptansätze.

2.6.1 Modellieren mit Keyframes

Bei der Modellierung mit Keyframes werden die Zwischenbilder (*in-betweens*) grundsätzlich aus Anfangs- und Endzustand (*keyframe*) des Objektes interpoliert, wobei zusätzliche Keyframes eingestreut werden können, um die Interpolation zu verfeinern. Die Keyframes müssen sich weder in der Form, noch in der Position gleichen. Dabei müssen die Stützstellen vorgegeben werden, damit die Interpolation korrekt ausgeführt werden. Die Interpolation kann linear oder kubisch sein oder mit Hilfe von Splines geschehen. Bei Keyframe-Animationen wird grundsätzlich zwischen den zwei Verfahren *image-based keyframe animation* und *parametric keyframe animation* unterschieden.

Image-based Keyframe Animation, Shape Interpolation Die Definition der Stützstellen wird durch Objekte (Körper) definiert. Das Hauptproblem der *Shape Interpolation* ist nicht die Interpolationsfunktion, sondern das Finden einer korrekten Beschreibung des Überganges zwischen zwei Objekten. Dieses Problem ist in 3D viel komplexer als in 2D, da sich Objekte nicht nur in der Anzahl Eckpunkte, sondern auch in der Anzahl Flächen unterscheiden. In einem ersten Schritt muss eine Zuordung gefunden werden, die einerseits aus der Zuordung der Flächen und anderseits aus der Zuordnung der Eckpunkte besteht. Mit einer Zuordnung wird dann die Übergangsfunktion der einzelnen Eckpunkte bestimmt. In einem zweiten Schritt wird mit Hilfe der Übergangsfunktion eine Interpolation ausgeführt, die dann für jede einzelne Fläche punktweise geschieht. Die *Shape Interpolation* eignet sich sehr gut für Körper-Metamorphosen (Abb. 9), hat aber die Nachteile, dass für realistische Transformationen viele Keyframes benötigt werden und dass das Modellieren etwas unflexibel ist. Der Rechenaufwand hängt von der Anzahl der Eckpunkte und der Interpolationsmethode ab.

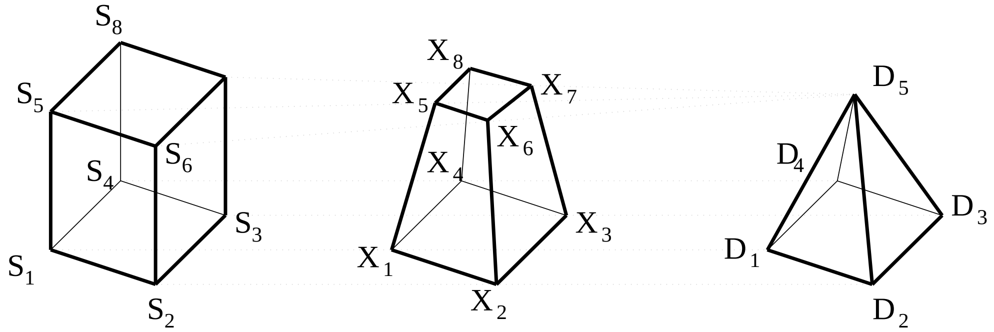


Abbildung 9: Shape Interpolation [MTT90].

Parametric Keyframe Animation, Parameter Interpolation Die Stützstellen der *Parameter Interpolation* werden durch Parameter der Objekte (Körper) definiert. Mit der Interpolationsmethode wir die Interpolation der Parameter bestimmt. Die Objekte in den *In-betweens* werden anhand der interpolierten Parameter transformiert. Die Definition der Stützstellen ist kurz und einfach und lässt einen gewissen kreativen Spielraum offen. Nachteilig ist, dass die Qualität stark von der richtigen und nicht immer einfachen Wahl der Parameter abhängt. Die Interpolation sollte so gewählt werden, dass ein möglichst kontinuierlicher Verlauf der Kurve im Raum und in der Zeit entsteht. Normalerweise werden C^2-stetige Kurven benutzt um beispielsweise bei Bewegungen im Raum die Ecken der Stützstellen zu glätten. Der Rechenaufwand hängt von der Anzahl Parameter und der Interpolationsmethode ab.

2.6.2 Algorithmische Animation

Die Transformationen werden algorithmisch beschrieben. Dazu werden physikalische Gesetze verwendet, um eine Transformation zu modellieren. Die einzelnen Transformationen haben Parameter, die sich im Verlauf der Zeit verändern können und durch Gesetze modelliert werden. Die Definition der Gesetze reicht von einfachen analytischen Funktionen bis zu hochkomplexen Prozessen. In der algorithimschen Animation wird bei der Modellierung der Transformationen zwischen den Ansätzen mit Kinematik und denjenigen mit Dynamik unterschieden.

Kinematic Algorithmic Animation Mit der Kinematik wird die Position und die Geschwindigkeit der Objekte beschrieben. Dieser Ansatz deckt etwa alle Bewegungsarten von Objekten, die mit 3D-Transformationen beschrieben werden können ab. Scherung und Skalierung werden auch dazugezählt, weil die Deformation nichts anderes ist als die unabhängige Verschiebung von Punkten eines Objektes. Die Modellierung gestaltet sich sehr flexibel, nur ist die Suche nach realistischen Gesetzen nicht immer einfach (siehe Pendelbewegung (13) auf Seite 51). Die Formel (1) beispielsweise beschreibt die Elongation eines Pendels auf analytische Weise. Der Rechenaufwand ist relativ klein und hängt nur von der Auswertung des Gesetzes (Formel) ab.

$$\alpha = A\sin(\omega t + \phi) \tag{1}$$

Dynamic Algorithmic Animation Dieser realistischere und komplexere Ansatz erlaubt die Berücksichtigung von physikalischen Gesetzen, die die Kinematik steuern. Auf die Objekte wirken Kraft und Drehmoment und beeinflussen Ort und Form der Objekte. Diese Art von Modellierung ist sehr realitätsnah, weil die Bewegungen und Deformationen mit Gesetzen der Natur beschrieben werden können. Dieser Ansatz eignet sich aber schlecht für längere Animationen, da sie eine grossen Rechenaufwand mit sich bringen.

2.7 Methoden der Animationskontrolle

Aus einer Animation, die aus 3D-Objekten und Transformationen besteht, können noch keine vernünftigen Einzelbilder gewonnen werden, weil weder aus den Objekten noch aus den Transformationen der Einsatz (zeitlich) der Transformationen hervorgeht. Hier greift nun die Animationskontrolle ein und kontrolliert die Einsätze der Transformationen. Damit ist es möglich aus der Beschreibung der Transformationen die Bewegungen oder Deformationen der Objekte in einem bestimmten Bild zu berechnen. Eine weitere Motivation der Animationskontrolle ist die Wiederverwendbarkeit von Transformationen. Es ist zwar prinzipiell möglich, ohne solche Methoden auszukommen, jede einzelne Transformation muss aber implementiert werden, auch wenn sie sich nur um eine Verschiebung in der Zeit unterscheiden.

2.7.1 Explizite Kontrolle (*Full Explicit Control*)

Die Beschreibung der Animation enthält alle Transformationen, die in der Szene vorkommen. Die Beschreibung entspricht einem genauen Einsatzplan der Transformationen, dabei können Veränderungen der Objekte — falls nötig für jedes einzelne Frame — explizit angegeben werden. Der Ansatz eignet sich für Key-Frame Animationen und für das interaktive Editieren von Animationen.

2.7.2 Algorithmische Animation (*Algorithmic Animation*)

Die algorithmische Animation kennt zwei Methoden der Animationskontrolle. Eine Animation kann mit *Inverse Dynamic* bzw. *Inverse Kinematic* ode mit *Forward Dynamic* bzw. *Forward Kinematic* kontrolliert werden. Unter *Forward* wird vorausgesetzt, dass die kinematischen, bzw. dynamischen Eigenschaften der Objekte bekannt sind. Das Objekt wird von einem Startzustand aus transformiert. Das zu animierende Objekt kennt Beginn und Ende der Transformation. Das Ende der Transformation kann auch mit einem kinematischen oder dynamischen Zielwert definiert werden. Im Gegensatz dazu wird unter *Inverse* ein gewünschter Endzustand des Objektes angegeben. Das zu animierende Objekt wird dann in den gewünschten Endzustand gebracht. Dabei werden die kinetischen bzw. dynamischen Gesetze eingehalten. Mit dieser Methode können sehr einfach realistische Animationen beschrieben werden. Diese Methode bringt für komplexe Objekte, speziell im dynamischen Fall, einen sehr grossen Rechenaufwand mit sich. Die Transformation in einen Endzustand hat auch den Nachteil, dass sie bei komplexen Objekten nicht immer eindeutig ist.

2.7.3 Adaptive Animationskontrolle (*Constraint-Based System*)

In der adaptiven Animationskontrolle werden die Transformationen der Objekte durch die Objekte selbst beeinflusst. Während der Berechnung der einzelnen Frames müssen die Zustände der einzelnen Objekte allen andern zur Verfügen stehen. Das Berücksichtigen von Hindernissen (*constraints*) beinhaltet eine Kollisionsabfrage der Objekte untereinander. Die Kollision eines beweglichen Objektes mit einem statischen Objekt lässt sich vernünftig lösen. Die Abfrage einer Kollision zwischen zwei beweglichen Objekten ist dagegen nicht immer trivial und der Rechenaufwand verhält sich quadratisch zur Anzahl der beweglichen Objekte. Deshalb werden die Objekte in zwei Klassen eingeteilt. Die eine Klasse enthält alle statischen Objekte, wie Hintergrund und Dekoration, die andere Klasse besteht aus allen animierten Objekten. So kann der Aufwand minimiert werden.

Damit ist die Einführung über Computer-Animation abgeschlossen. Das nun folgende Kapitel befasst sich mit der Implementation eines Ansatzes und erklärt das Zusammenspiel von BOOGA und der Computer-Animation.

3 Computer-Animation für BOOGA

In diesem Kaptitel zeige ich , wie ich meinen selbstentwickelten Ansatz implementiert und wie ich die Animation in BOOGA eingebettet habe. Dazu werde ich zuerst den gewählten Ansatz erklären und später auf die Implementation der Klassen eingehen. In diesem Abschnitt werden auch die nötigen Begriffe eingeführt und eine genaue Beschreibung der Animationskonstrukte gemacht.

3.1 Begriffe

Transformation	:	Allgemeine Transformation in R^3.
Aktion	:	Transformation mit Parametern der Animationskontrolle und der Modellierung.
Modellierungs-Funktion	:	Bestimmt das dynamische, zeitliche Verhalten der Transformation.
Animations-kontrolle	:	Kontrollmethode für den Einsatz einer Transformation.
Zielwert	:	Zielwert der Transformation.
frame	:	Variable und Dimension der Zeit, $\in R$, normalerweise in Sekunden.
startFrame	:	Beginn einer Aktion.
endFrame	:	Ende einer Aktion.
[start, end]	:	Definitionsintervall der Modellierungsfunktion.
step	:	Stufenlänge der Treppenfunktion (Formel (10) Seite 37).
value	:	Funktionswert der Modellierungsfunktion.
ticks	:	Funktionswert der Modellierungsfunktion, Wiederholungen mitberücksichtigt.

3.2 Der Ansatz

Die direkte Wahl eines Ansatzes stand nie an erster Stelle, da meine Zielsetzungen mehr die Einbettung in BOOGA gewichteten. Eine schlechte Einbettung wäre auch im Widerspruch zum Gebot der Wiederverwendung gestanden, weshalb sich der vorliegende Ansatz nach und nach herauskristallisierte.

Anfangs versuchte ich, mir mit einfachen und festprogrammierten Manipulationen von 3D-Objekten, einen Einblick in den Aufbau von BOOGA zu verschaffen. Auch wenn ich grundsätzlich mit der Wahl frei war, stellte die Plattform BOOGA der Entwicklung einige Grundbedingungen. Einerseits musste der Ansatz objektorientiert implementierbar sein und anderseits musste ich einen Ansatz entwickeln, der nicht nur statische Eingabefiles verträgt. Die Trennung von Animationskonstrukten mit der Szene musste ich schon früh verwerfen, da dynamische Änderungen in der Szene unvorhersehbare Folgen in der Animationbeschreibung haben können. Diese Seiteneffekte sind nur mit aufwendigen Protokollen umgehbar.

Die ersten Transformationen von Objekten habe ich mit einer eigenen Klasse `Anim` von `Transformer3D` abgeleitet, realisiert. Die Transformationen wurden beim Traversieren des Szenenbaumes ausgeführt. Dazu musste ich `visit`-Methoden für `Primitive3D` und davon

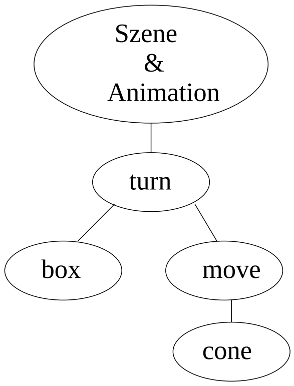


Abbildung 10: Animation und Szene vereint.

abgeleitete Objekte implementieren. Die einzelnen Methoden enthielten eine bestimmte Transformation. Mit der `apply`-Methode wurde das nächste Bild berechnet, indem eine Traversierung ausgeführt wurde. Dieser Ansatz war für Animationen unbrauchbar, aber für das Kennenlernen von *BOOGA* wertvoll. Mein Betreuer gab mir dann eine grobe Einbettung vor, die für Animationsobjekte eine Ableitung von `Object3D` vorsah.

Mit der Einbettung, die eine Vereinigung der Animation und der Szene (Abb. 10) vorsah, verwarf ich Keyframes, weil dieser Ansatz in *BOOGA* eine aufwendige und komplizierte Speicherungstechnik für die einzelnen Keyframes mit sich gebracht hätte. Ein vollständiger OO-Ansatz war zu Beginn der Arbeit unrealistisch, deshalb konzentrierte ich mich auf einen Ansatz in Richtung algorithmische Animation. Zu Beginn war mir aber noch nicht klar, wie ich die Animationsobjekte ansprechen sollte. Eine Möglichkeit war, die Animationsobjekte mit Events in Bewegung zu setzen, wie beispielsweise in *The Inventor Mentor* [Wern94]. Ich entschied mich für eine direkte Ansteuerung der Animationsobjekte, bei der man den Animationsobjekten den gewünschten Zeitpunkt (frame-Wert) übergibt und die Animationsobjekte die zu animierenden Objekte in den gewünschten Zustand bringen. So kann ein Bild für einen beliebigen Zeitpunkt berechnet werden. Die Modellierung der Transformationen habe ich wie in der algorithmischen Animation mit einer Auswahl von Funktionen (Gesetzen) ausgestattet.

3.2.1 Neue Objektklassen in BOOGA

Im Verlauf dieses Projektes wurden folgende Klassen für die Realisierung der Animation auf *BOOGA* erstellt:

- `ActionInfo`. Klasse der Aktionen.
- `ActionInfoAttr`. Attribute der Aktionen. Wird anhand von `ActionInfoAttr.specifierAttr` automatisch generiert.
- `Animation3D`. Basisklasse der Animationsobjekte.
- `Animation3DAttr`. Attribute der Animationsobjekte. Wird anhand von `Animation3DAttr.specifierAttr` automatisch generiert.

- `AnimationFunction`. Basisklasse der Modellierungsfunktionen.
- `Grow3D`. Animationsobjekt der Skalierung.
- `Move3D`. Animationsobjekt der Translation.
- `Shear3D`. Animationsobjekt der Scherung.
- `Tumble3D`. Animationsobjekt der allgmeinen Bewegung im Raum.
- `TumbleCenter3D`. Attribut der Aktionen von `tumble`.
- `TumbleDirection3D`. Attribut der Aktionen von `tumble`.
- `TumblePath3D`. Attribut der Aktionen von `tumble`.
- `Turn3D`. Animationsobjekt der Rotation.

3.3 Aufbau des Animationskonstrukts

Das Animationskonstrukt besteht zum einen aus einer Methode zur Kontrolle der Animation und zum anderen aus der Modellierung der Transformation. Mein Ansatz geht davon aus, dass ein Animationsobjekt das zu animierende Objekt und mehrere gleichartige Transformationen (hier Aktionen genannt) aufnimmt und die Transformationen entsprechend ausführen kann. Da ein Animationsobjekt mehrere Transformationen aufnimmt, kann eine Transformation dekomponiert und aus einfacheren Transformationen zusammengesetzt werden. Das zeitliche Überlappen von Transformationen ist in jeder Form erlaubt. Ein Animationskonstrukt sieht wie folgt aus:

```
Transformationsart {
  Aktion1 (Animationskontrolle) {
    Modellierung einer Transformation
  }
  Aktion2 (Animationskontrolle) {
    Modellierung einer Transformation
  }
  ...

  Ein-/Ausschalten des Animationsobjektes
  3D-Objekt
}
```

Bei der Transformationsart wird zwischen den fünf 3D-Transformationen `grow` (Skalierung), `move` (Translation), `shear` (Scherung), `tumble` (allgemeine Bewegung in Raum) und `turn` (Rotation) unterschieden. Die Animationskontrolle bestimmt den Einsatz der Aktion durch Start- und Endframe. Optional kann eine Aktion wiederholt werden und zwischen den Wiederholungen kann ein Pause definiert werden. Die Modellierung der Transformation wird mit einem Zielwert, einer Modellierungsfunktion und ihren Parametern beschrieben.

3.4 Der Parser

Für das Einlesen von Szenen, die in der Metasprache BSDL geschrieben sind, wird ein Parser verwendet. Beim Einlesen der Datei wird die Szene im Speicher aufgebaut. Der BSDL-Parser ist sowohl statisch als auch dynamisch erweiterbar. Die statische Erweiterung geschieht durch Anpassen der Initialisierungsfunktionen des Parsers. Der Parser kennt auch die Möglichkeit des dynamischen Hinzufügens von neuen Exemplaren und Attributen. Exemplare sind für Objekte zuständig und Attribute bestimmen die Eigenschaften des Objektes.

Beim Einlesen einer BSDL-Datei wird für jeden einzelnen Bezeichner überprüft, ob dieser registriert worden ist. Ist der Bezeichner ein Objekt, wird ein leeres Exemplar mit der Memberfunktion `make` erzeugt. Die Parameter, die zum Objekt gehören, werden der Memberfunktion `make` des Objektes übergeben, damit eine korrekte Initialisierung mit den Parametern geschehen kann. Falls der Bezeichner ein Attribut war, wird ein Attribut-Objekt erzeugt und mit den entsprechenden Parametern initialisiert. Danach wird mit der Memberfunktion `setAttribute` der entsprechednen Objektklasse das Attribut im Objekt gesetzt.

Der Parser wurde um folgende Bezeichner erweitert und in `BSDLParserInit.C` eingetragen:

Erweiterung des Parsers		
Objekt	Attribute	Basis-Attributklasse
`grow`	`on`, `off`	Animation3DAttr
`move`	`on`, `off`	Animation3DAttr
`tumble`	`on`, `off`	Animation3DAttr
`turn`	`on`, `off`	Animation3DAttr
`shear`	`on`, `off`	Animation3DAttr
`action`	`alpha`, `axis`, `center`, `direction`, `scalefactor`, `shearfactor`, `tumblecenter`, `tumbledirection`, `tumblepath`	ActionInfoAttr

Tabelle 1: Erweiterung des Parsers.

3.5 Animationsobjekte (`Animation3D`)

`Animation3D` ist von `Object3D` abgeleitet (Abb. 11). `Animation3D` übernimmt als oberste Instanz und Schnittstelle zum Anwender die Interpretation und Ausführung aller Transformationen. Die Ausführung geschieht über die Memberfunktion `frame` mit dem Übergabeparameter `frame` (aktuelles Frame, frame-Wert). Anhand des frame-Wertes wird das zu animierende Objekt in den gewünschten Zustand transformiert. Falls sich seit dem letzten Aufruf nichts geändert hat, wird `false` zurückgegeben, sonst `true`. Mit den Memberfunktionen `turnOn` und `turnOff` wird das Animationsobjekt ein- bzw. ausgeschaltet. Der

Zustand kann mit `isOn` abgefragt werden. Die Zuweisung einer Aktion zu einem Animationsobjekt geschieht mit `adoptAction`. `adoptAction` ordnet intern die Aktionen aufsteigend nach dem Parameter `startframe`. Bei Gleichheit entscheidet die Eingabereihenfolge. Die Ordnung der Aktionen bestimmt die Reihenfolge der Transformationen.

Klassendiagramm

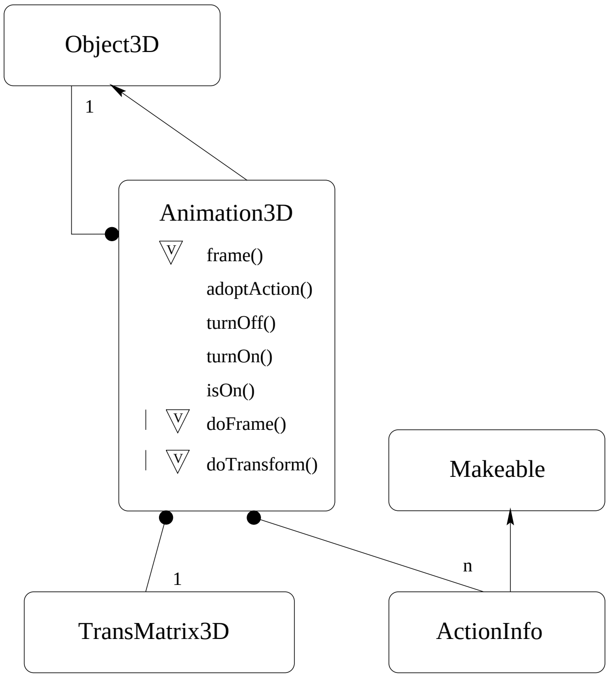


Abbildung 11: `Animation3D`.

3.5.1 Skalierung (`Grow3D`)

`Grow3D` (Abb. 12) übernimmt die Funktionalität der Skalierung von Objekten. Mit dem Skalierungsvektor lässt sich jedes beliebige Objekt in allen drei Dimensionen individuell skalieren. Die Skalierung ist kommutativ. Die Kommutativität erlaubt beliebiges Überlagern von Skalierungen, ohne dass unangenehme Seiteneffekte entstehen. Die Überlagerungsmöglichkeit lässt beispielsweise eine separate Modellierung aller drei Dimensionen zu. Weiter beinhaltet `Grow3D` auch die Möglichkeit der Spiegelung von Objekten, wozu man die zwei Skalierungsfaktoren 1 und einen -1 gleichsetzt.

Klassendiagramm

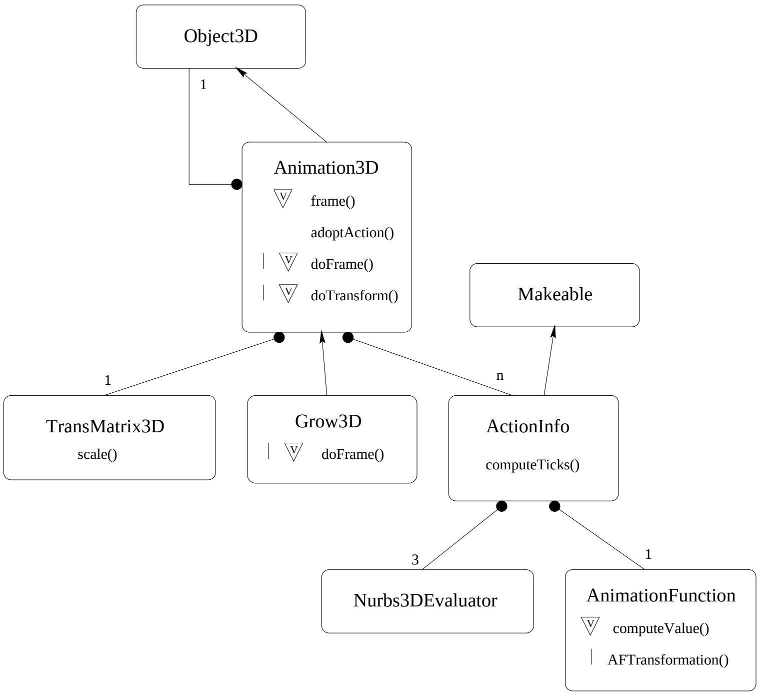


Abbildung 12: `Grow3D`.

Definition der Skalierung Die Skalierung wird mit dem Skalierungsvektor $(s_x, s_y, s_z) \in R^3$ und dem Zentrum $(0,0,0)$ durch folgende Matrix definiert:

$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2}$$

Sprachdefinition Die Skalierung im Raum R^3 wird mit dem Schlüsselwort `grow` eingeleitet. Die Skalierung wird mit einer Aktion (`action`) zeitlich kontrolliert. Die Modellierung der Skalierung geschieht mit der Angabe des Skalierungsvektors (s_x, s_y, s_z) und der Modellierungsfunktion mit ihren Parametern. Optional kann das Skalierungszentrum angegeben werden, sonst wird das Zentrum der Boundingbox des zu animierenden Objektes angenommen.

```
grow {
  action (startframe, endframe, times, wait) {
    scalefactor ([x, y, z], "myFunction", start, end, step);
    center ([x, y, z]);
  }
  myAnimatedObject;
}
```

Interaktionsdiagramm
Eingabebeispiel des Objektes `t` des Interaktionsdiagrammes (Abb. 13):

```
grow {
  action (2, 7) {
    scalefactor ([1, 2, 3], "id", 0.1, 0.8, 1/4);
    center ([13, 19, 17]);
  }
  ball;
}
```

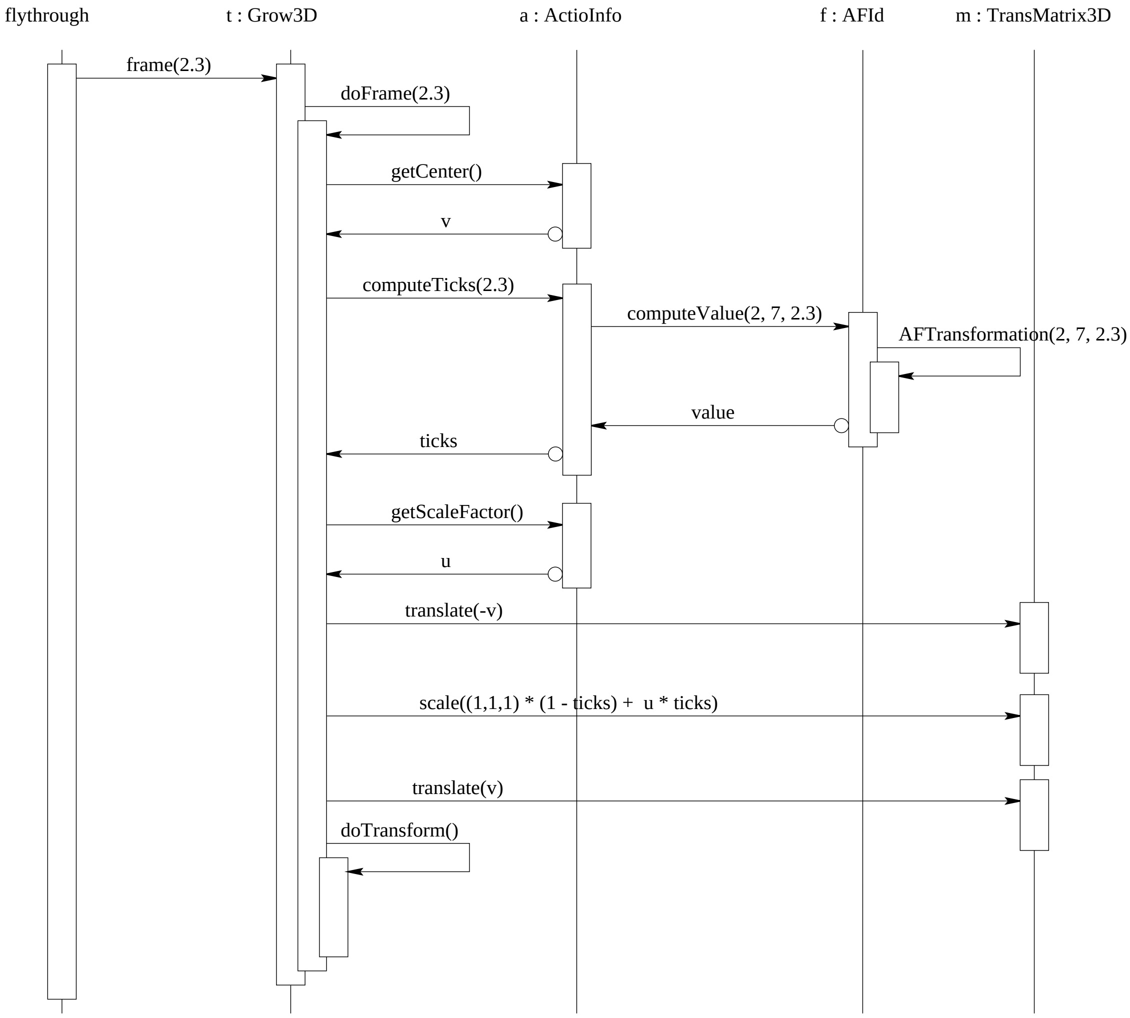


Abbildung 13: Interaktionsdiagramm von `Grow3D`.

3.5.2 Translation (`Move3D`)

`Move3D` (Abb. 14) erlaubt eine beliebige, gradlinige Translation der Objekte im Raum. Weil auch die Translation zu den einfacheren Transformationen gehört, besitzt sie die angenehme Eigenschaft der Kommutativität. Damit sind auch bei Translationen beliebige Überlagerungen möglich.

Klassendiagramm

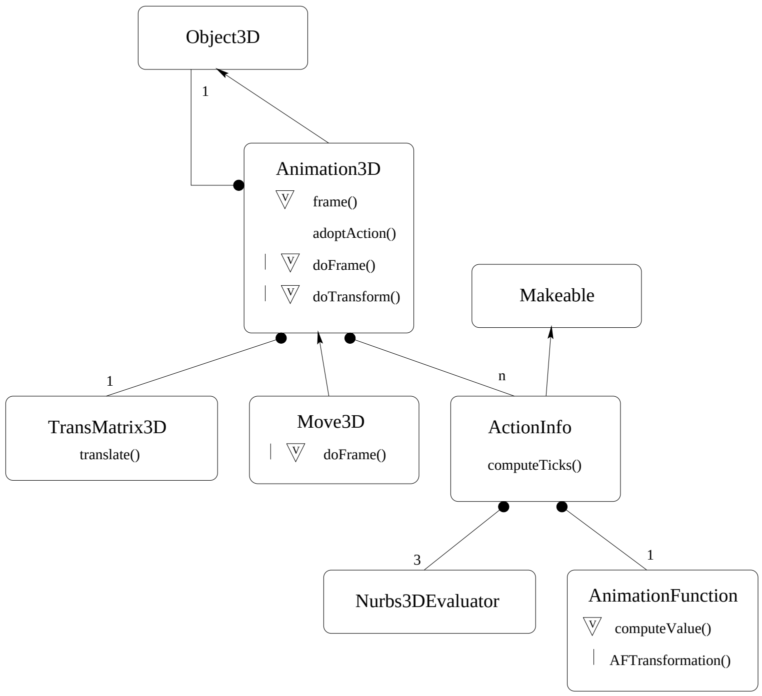


Abbildung 14: `Move3D`.

Definition der Translation Die Translation wird mit dem Translationsvektor $(d_x, d_y, d_z) \in R^3$durch folgende Matrix definiert:

$$T(d_x, d_y, d_z) = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (3)$$

Sprachdefinition Die Translation im Raum R^3 wird mit dem Schlüsselwort `move` eingeleitet. Die Translation wird mit einer Aktion (`action`) zeitlich kontrolliert. Die Modellierung der Translation geschieht mit der Angabe des Translationsvektors $(d_x, d_y, d_z) \in R^3$

und der Modellierungsfunktion mit ihren Parametern.

```
move {
  action (startframe, endframe, times, wait) {
    direction ([x, y, z], "myFunction", start, end, step);
  }
  myAnimatedObject;
}
```

Interaktionsdiagramm
Eingabebeispiel des Objektes `t` des Interaktionsdiagrammes (Abb. 15):

```
move {
  action (5, 9.1) {
    direction ([1, 2, 3], "step", 0, 0.8, 1/7);
  }
  action (2, 7) {
    direction ([1, 1, -37], "saw", 0.3, 1, 1/7);
  }
  ball;
}
```

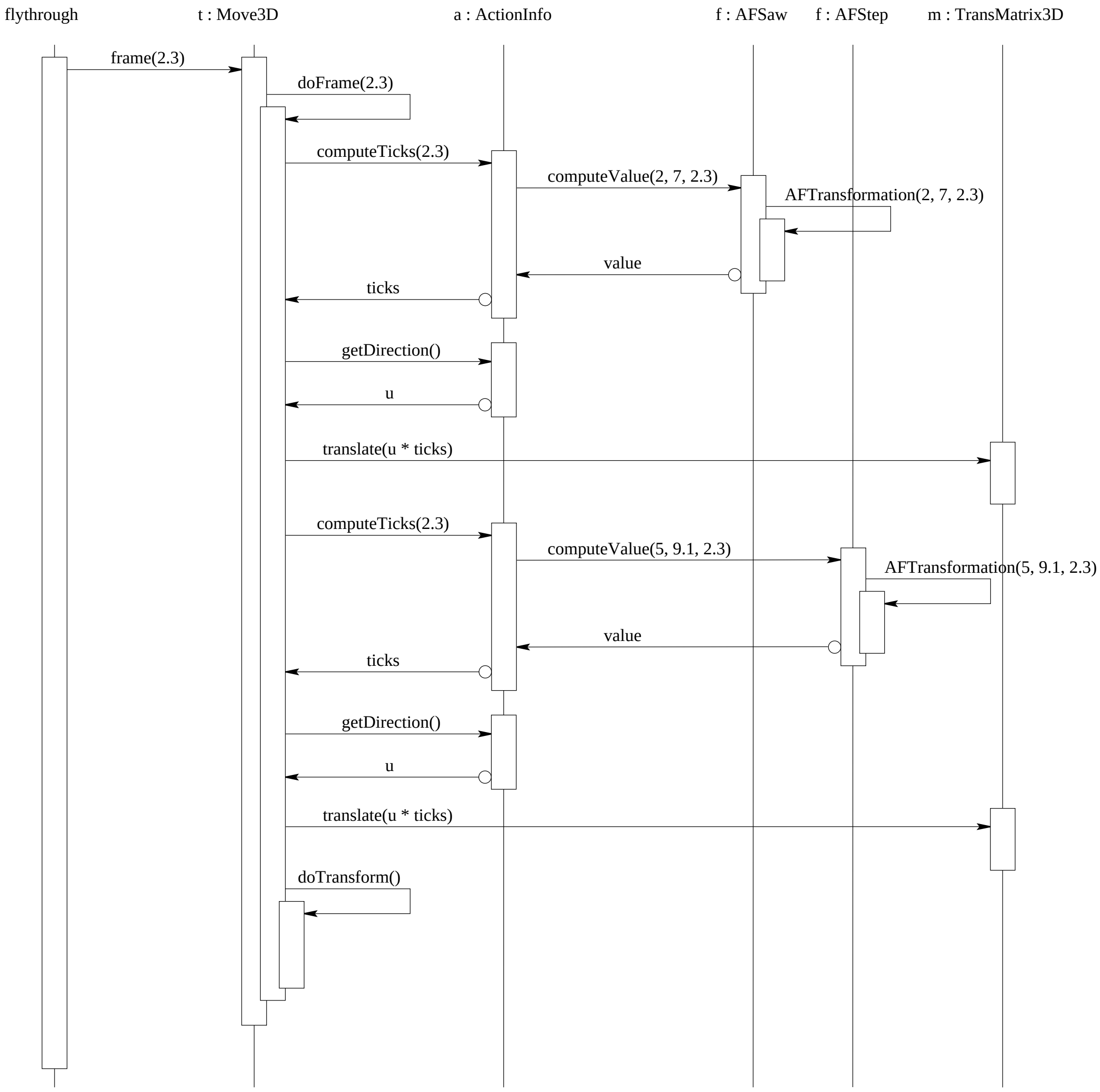


Abbildung 15: Interaktionsdiagramm von `Move3D`.

3.5.3 Scherung (`Shear3D`)

`Shear3D` (Abb. 16) erlaubt eine Scherung in der Ebene. Die Scherung wird durch einen Normalvektor der Scherungsebene, ein Zentrum und einem Scherungsvektor beschrieben. Durch die Angabe des Normalvektors wird eine Rotation definiert, die die Scherungsebene in die xy-Ebene dreht. Die Rotationsachse wird aus dem Vektroprodukt von Normalvektor und z-Achse gewonnen. Das Rotationszentrum wird dem Scherungszentrum gleichgesetzt.

Klassendiagramm

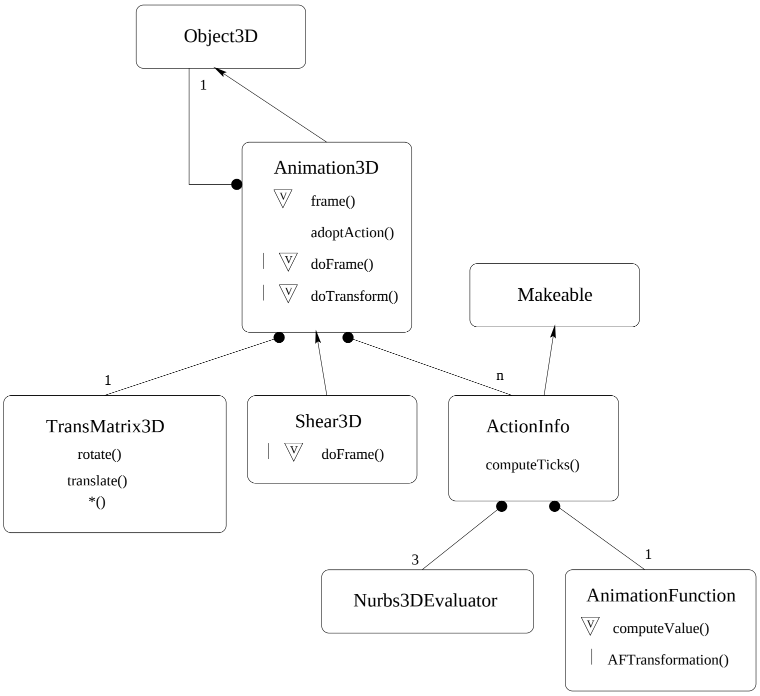


Abbildung 16: `Shear3D`.

Definition der Scherung Die Scherung in der xy-Ebene mit dem Scherungsvektor $(s_x, s_y) \in R^2$ und Zentrum $(0,0,0)$ wird durch folgende Matrix definiert:

$$SH_{xy}(sh_x, sh_y) = \begin{pmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4}$$

Sprachdefinition Die Scherung in der Ebene wird mit dem Schlüsselwort `shear` eingeleitet. Die Scherung wird mit einer Aktion (`action`) zeitlich kontrolliert. Die Modellierung der Scherung geschieht mit der Angabe des Scherungsvektors (sh_x, sh_y), der Modellierungsfunktion mit ihren Parametern und dem Normalvektor der Scherungsebene. Optional kann das Scherungszentrum angegeben werden, sonst wird das Zentrum der Boundingbox des zu animierenden Objektes angenommen.

```
shear {
  action (startframe, endframe, times, wait) {
    shearfactor ([x, y],"myFunction", start, end, step);
    axis ([x, y, z]);
    center ([x, y, z]);
  }
  myAnimatedObject;
}
```

Interaktionsdiagramm
Eingabebeispiel des Objektes `t` des Interaktionsdiagrammes (Abb. 17):

```
grow {
  action (2, 7) {
    shearfactor ([1, 3], "step", 0, 1, 0);
    center ([-13, 19, -31]);
    axis([0, 0, 1]);
  }
  ball;
}
```

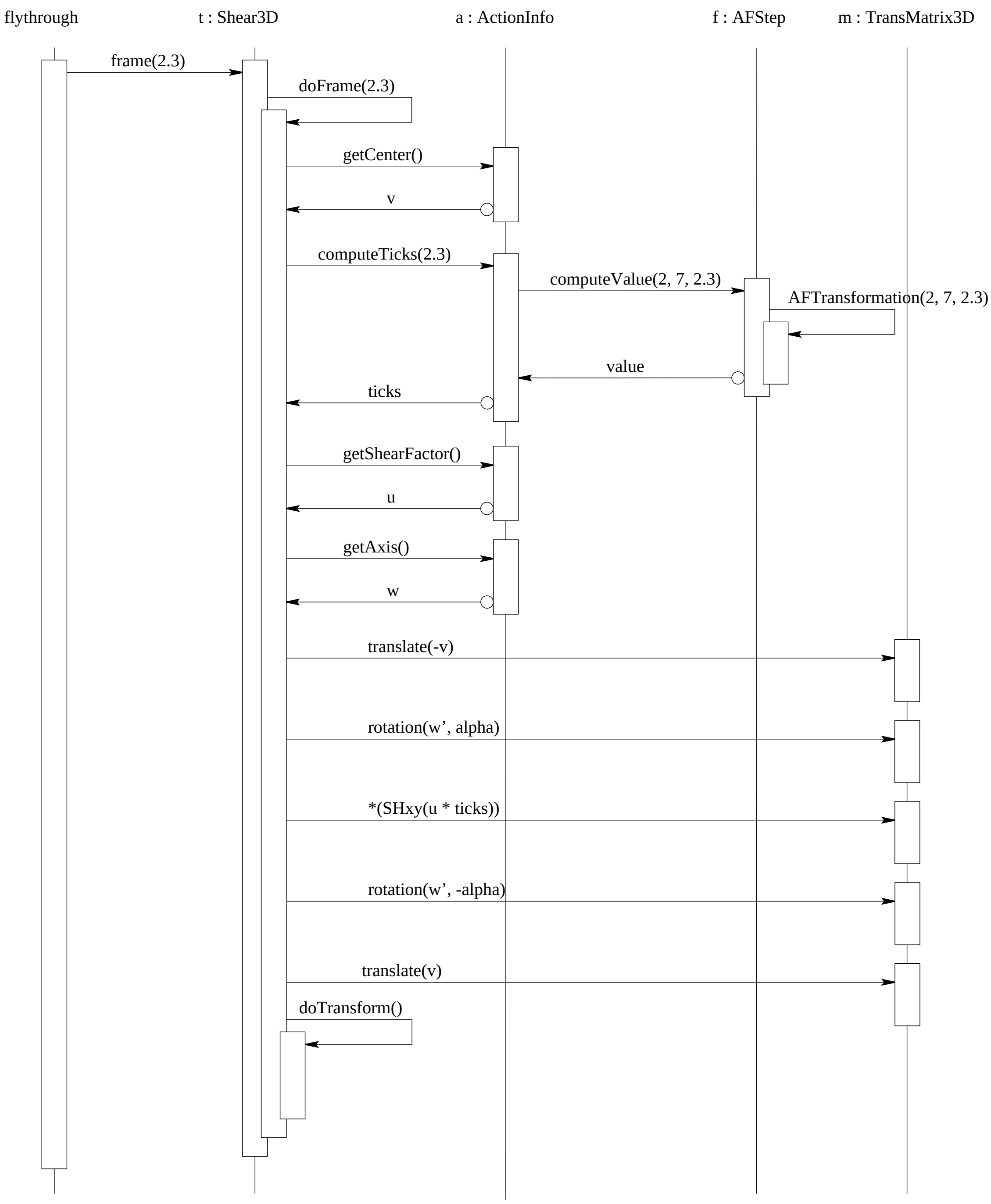


Abbildung 17: Interaktionsdiagramm von `Shear3D`.

3.5.4 Allgemeine Bewegung im Raum (`Tumble3D`)

`Tumble3D` (Abb. 20) erlaubt eine allgemeine Bewegung im Raum, die aus einer Translation und einer Rotation mit modellierbarem Rotationszentrum besteht. Die Rotation wird hier als Ausrichtung des Objektes interpretiert.

Mit der NURBS in `tumblepath` (Abb. 18) wird die Translation beschrieben. Der Translationsvektor wird bezüglich dem Startpunkt ($u = 0$) der Kurve berechnet, d.h. eine Translation der Kurve verändert deren Verhalten nicht.

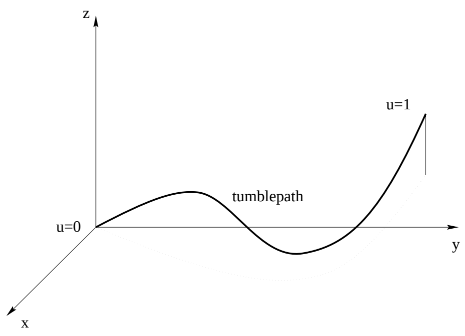


Abbildung 18: Translationspfad.

Das Rotationszentrum wird mit der NURBS in `tumblecenter` (Abb. 19) beschrieben. Das Zentrum bezieht sich auf das zu animierende Objekt. Damit ist eine beliebige Modellierung des Rotationszentrums möglich. Der Vektor, definiert durch die NURBS in `tumblecenter` und `tumbledirection`, gibt die Ausrichtung des Objektes an. Die Ausrichtung geschieht relativ zum Startvektor ($u = 0$).

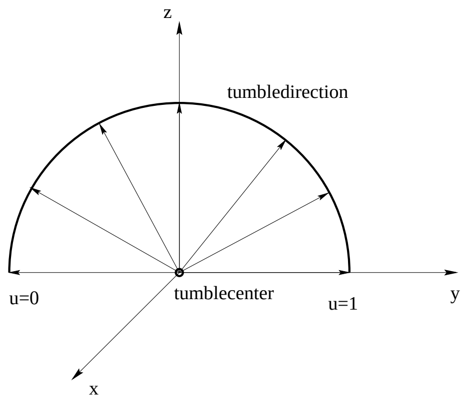


Abbildung 19: Ausrichtung des Objektes.

Klassendiagramm

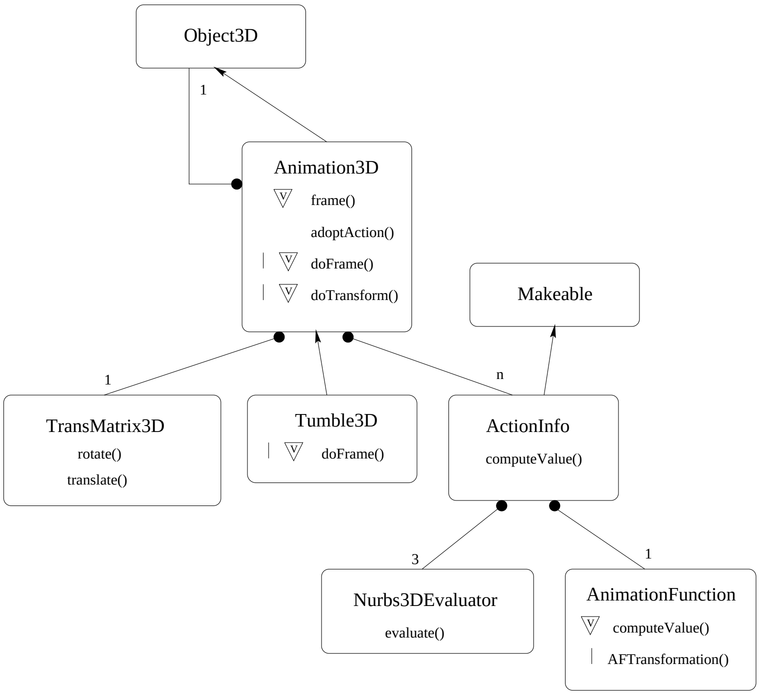


Abbildung 20: `Tumble3D`.

Sprachdefinition Die allgemeine Bewegung im Raum wird mit `tumble` eingeleitet. Die Bewegung und die Ausrichtung des Objektes wird mit einer Aktion (`action`) zeitlich kontrolliert. Die Translation wird mit einer Modellierungsfunktion mit ihren Parametern modelliert. Der Translationspfad wird mit einer NURBS in `tumblepath` modelliert. Die Ausrichtung des Objektes ist optional und wird mit den zwei NURBS-Kurven in `tumblecenter` und `tumbledirection` definiert. Die Ausrichtung kennt keine Modellierungsfuktionen und verhält sich (dynamisch) wie der Translationspfad. Falls die beiden Kurven nicht korrekt sind, wird die Ausrichtung des Objektes ohne Fehlermeldung fallengelassen.

Die Eingabe der NURBS ist für `tumblepath`, `tumblecenter` und `tumbledirection` identisch. Die NURBS muss vom Typ Kurve sein, sonst wird sie nicht akzeptiert. Die NURBS-Kurve wird durch den Knotenvektor (`uknots`), dem Kontrollpolygon (`vectors`) und den Gewichten (`weights`) definiert. Das Array des Knotenvektors beginnt mit dem Grad p, gefolgt von U. Das Kontrollpolygon besteht aus der Anzahl der Kontrollpunkte und den Kontrollpunkten selbst. Die Eingabe der Gewichte besteht aus der Anzahl der Gewichte, gefolgt von den Werten. Für die Eigenschaften und die Implementation der NURBS sei hier auf die Arbeit von [Bächler95] verwiesen.

```
tumble {
  action (startframe, endframe, times, wait) {
    tumblepath ("myFunction", start , end, step) {
```

```
      nurbs "curve" {
        uknots (3,0,0,0,1,1,1);    \\ Grad 3
        vectors (4,[0, -300, 0],[20, -100, 0],[20, 100, 20],[20, 300, 20]);
        weights (4,1,1,1,1);
      }
    }
    tumblecenter {
      nurbs "curve" {
        uknots (3,0,0,0,1,1,1);
        vectors (4,[0, 0, 0],[0, 0, 0],[0, 0, 0],[0, 0, 0]);
        weights (4,1,1,1,1);
      }
    }
    tumbledirection {
      nurbs "curve" {
        uknots (3,0,0,0,1,1,1);
        vectors (4,[5, 0, 0],[0, 0, -5],[-5, 0, 0],[0, 0, 5]);
        weights (4,1,1,1,1);
      }
    }
  }
  myAnimatedObject;
}
```

Definition von NURBS-Kurven NURBS [Bächler95] (**N**on-**U**niform **R**ational **B**-**S**pline) basieren auf den B-Splines und sind eine Verallgemeinerung von Bézierkurven und B-Splines. Mit NURBS können fast alle nur erdenklichen (stetigen) Kurven modelliert werden. Die NURBS-Kurve

$$C(u) = \frac{\sum_{i=0}^{n} N_i^p(u)\omega_i P_i}{\sum_{i=0}^{n} N_i^p(u)\omega_i} \tag{5}$$

wird mit den B-Splines

$$N_i^p(u) = \frac{u - u_{i-1}}{u_{i+p-1} - u_{i-1}} N_i^{p-1}(u) + \frac{u_{i+p} - u}{u_{i+p} - u_i} N_{i+1}^{p-1}(u) \tag{6}$$

und

$$N_i^0 = \begin{cases} 1 & : \; u_i \leq u < u_{i+1} \\ 0 & : \; \text{sonst} \end{cases} \tag{7}$$

über dem Intervall $[a, b]$ definiert. Die B-Splines-Basisfunktionen vom Grade p werden durch den Knotenvektor $U = \{a, \ldots, a, u_p, \ldots, u_{n-1}, b \ldots, b\}$ definiert, wobei $a \leq u_p \cdots u_{n-1} \leq b$ gelten muss. P_i sind die Kontrollpunkte des Kontrollpolygons.

Interaktionsdiagramm

Eingabebeispiel des Objektes `t` des Interaktionsdiagrammes (Abb. 21):

```
tumble {
  action (2, 7) {
    tumblepath ([1, 2, 3], "id", 0, 1, 0) {...}
    tumblecenter {...}
    tumbledirection {...}
  }
  ball;
}
```

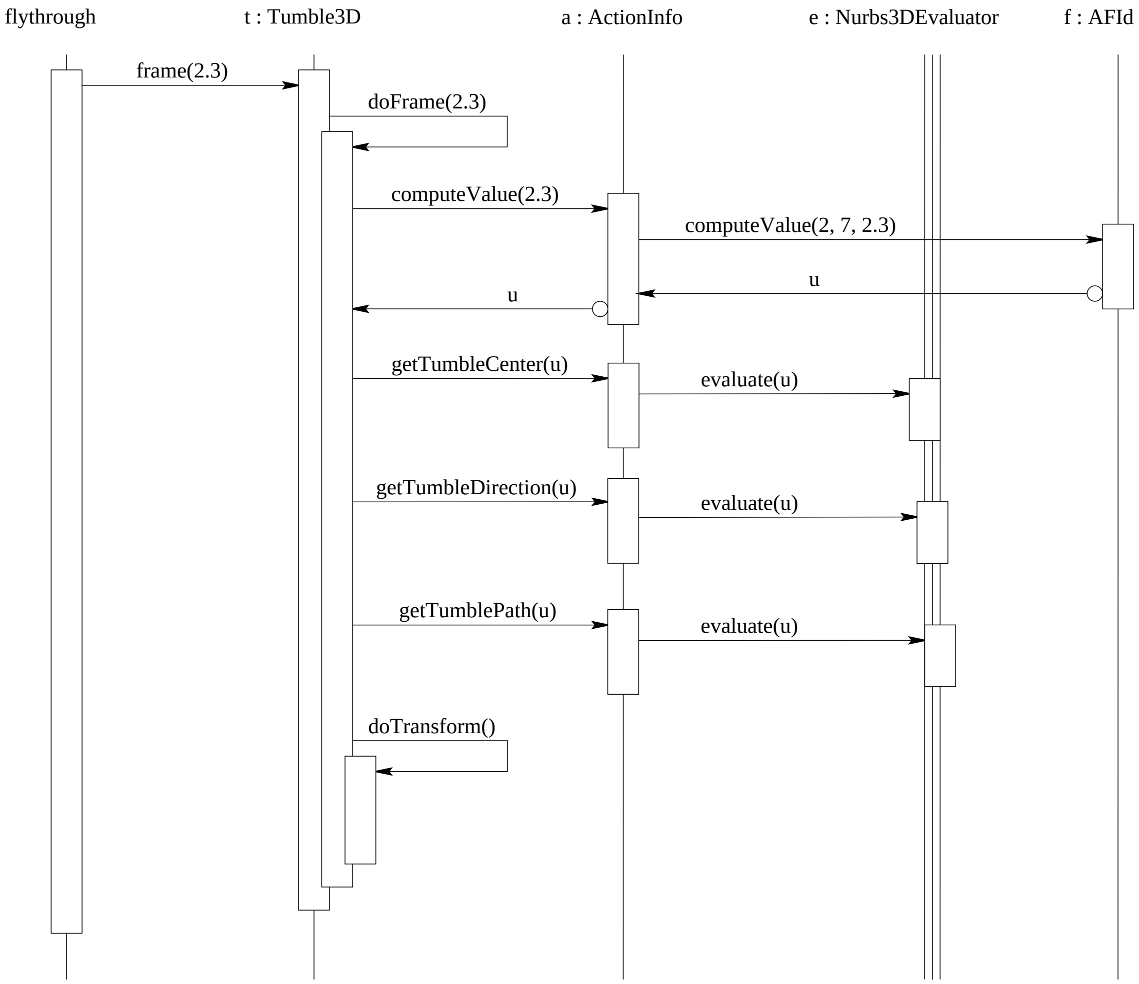


Abbildung 21: Interaktionsdiagramm von `Tumble3D`.

3.5.5 Rotation (`Turn3D`)

`Turn3D` (Abb. 22) bietet eine beliebige Rotation um eine bestimmbare Rotationsachse $(r_x, r_y, r_z) \in R^3$ mit Zentrumsangabe. Objekte können so in jede nur erdenkliche Lage gedreht werden. Die Rotation kommutiert nicht, weshalb die Reihenfolge der Aktionen, die die einzelnen Rotationen beschreiben, entscheidend ist.

Klassendiagramm

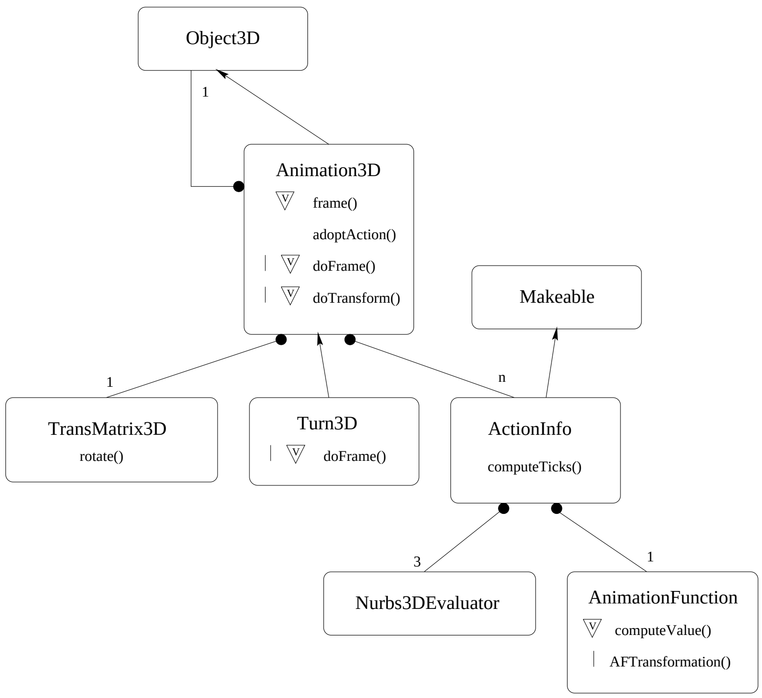


Abbildung 22: `Turn3D`.

Definition der Rotation Die Rotation um die z-Achse mit dem Rotationswinkel $\theta \in R$ und Zentrum $(0, 0, 0)$ wird durch folgende Matrix definiert:

$$R_z(\theta) = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 & 0 \\ sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{8}$$

Sprachdefinition Die Rotation im Raum wird mit dem Schlüsselwort `turn` eingeleitet. Die Rotation wird mit einer Aktion (`action`) zeitlich kontrolliert. Die Modellierung der Rotation geschieht mit der Angabe des Rotationswinkels θ [Grad], der Modellierungsfunktion mit ihren Parametern und der Rotationsachse $\in R^3$. Optional kann das Rotationszentrum angegeben werden, sonst wird das Zentrum der Boundingbox des zu animierenden Objektes angenommen.

```
turn {
  action (startframe, endframe, times, wait) {
    alpha (angle, "myFunction", start, end, step);
    axis ([x, y, z]);
    center ([x, y, z]);
  }
```

```
  myAnimatedObject;
}
```

Interaktionsdiagramm
Eingabebeispiel des Objektes `t` des Interaktionsdiagrammes(Abb. 23):

```
turn {
  action (2, 7) {
    scalefactor (180, "smoothstep", 0, 1, 0);
    axis ([0, 1, 1]);
  }
  ball;
}
```

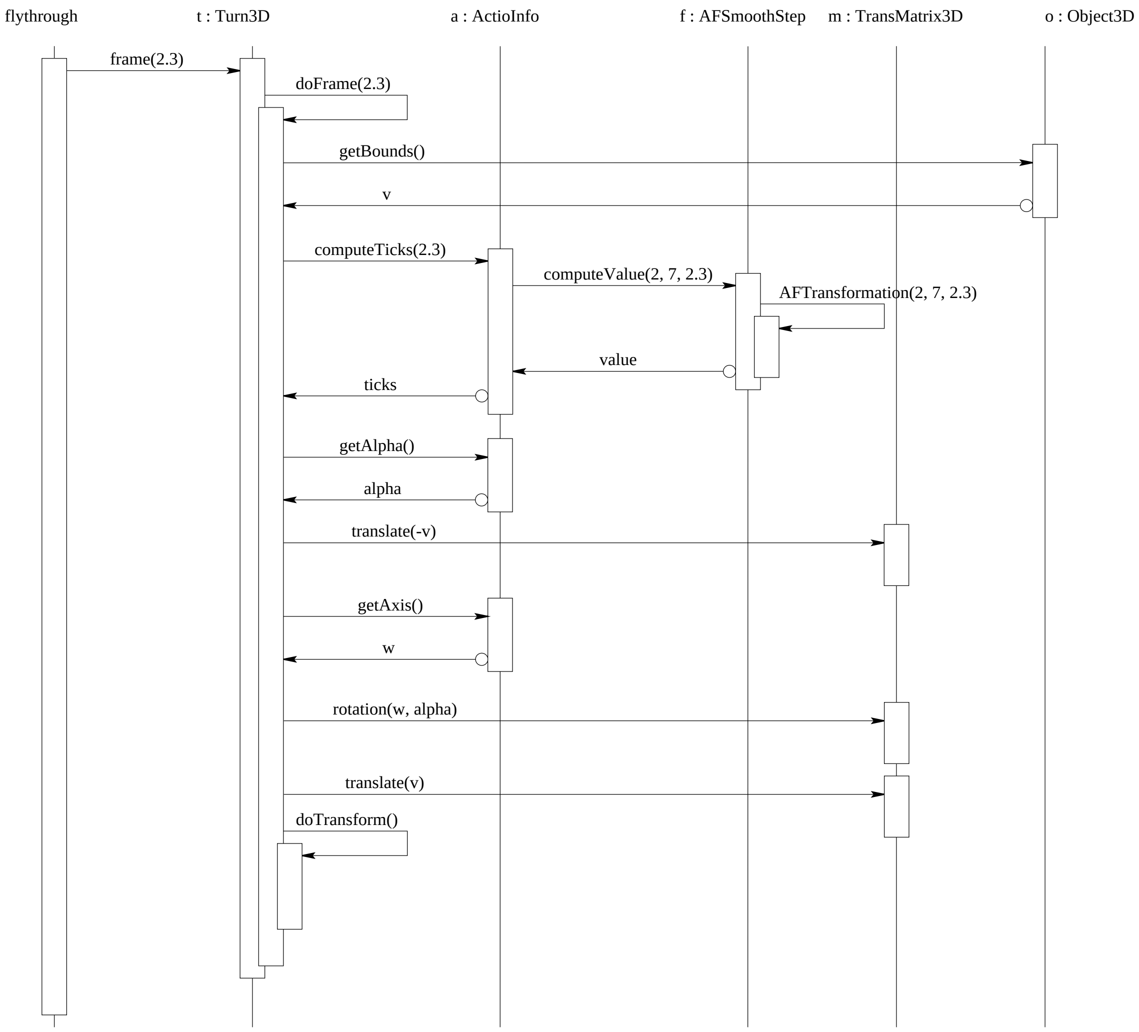


Abbildung 23: Interaktionsdiagramm von `Turn3D`.

3.6 Methoden der Animationskontrolle (`ActionInfo`)

Die einzelne Transformation wird mit den Parametern der Aktion kontrolliert. Mit der Angabe der Parameter werden Beginn (`startframe` $\in R$) und Ende (`endframe` $\in R$) bestimmt. Optional kann eine Transformation über den Parameter `times` $\in N^0$ wiederholt werden. Mit 0 wird eine unendliche Wiederholung eingeleitet, und beim Weglassen wird 1 als Defaultwert angenommen. Mit dem optionalen Parameter `wait` $\in R^+ \cup \{0\}$ kann zwischen den Wiederholungen eine Pause definiert werden. `startframe`, `endframe` und `wait` leben in der Dimension der Zeit.

$$computeTicks(x) = \begin{cases} times = 0 & : \text{ unendliche Wiederholung akkumulieren} \\ times = 1 & : computeValue(x) \\ times > 1 & : \text{ endliche Wiederholungen akkumulieren} \end{cases} \tag{9}$$

Bei der Auswertung einer Aktion wird die Memberfunktion `computeTicks` (Formel (9)) von `ActionInfo` aufgerufen, welche die Modellierungsfunktion auswertet. Dabei werden mögliche Wiederholungen und Pausen berücksichtigt. Die Wiederholung ist durch stetige Fortsetzung (Abb. 24, links) mit sich selbst definiert. Für den endlichen Fall wird entsprechend `times` in positiver Richtung fortgesetzt. Im unendlichen Fall wird auf beiden Seiten unendlich fortgesetzt. Die Pause ist als konstante (stetige) Fortsetzung der Modellierungsfunktion auf der rechten Seite definiert. Der Rückgabewert dient als Faktor bei der Berechnung der aktuellen Transformation. Die Abb. 24 zeigt die Intervalltransformation aus dem Raum der Modellierungsfunktion in den Raum, wo die Aktionen leben, anhand eines Eingabebeispiels (siehe weiter unten). Die konkrete Intervalltransformation wird von `computeValue` aus `AnimationFunction` ausgeführt.

Eingabebeispiel zu Abbildung 24

```
grow {
  action (5.4, 10.8, 2, 8.1) {
    scalefactor ([1, 2, 1], "id", 0.3, 0.8, 0);
    center ([0, 0, 0]);
  }
  ball;
}
```

3.7 Modellierung der Transformation (`ActionInfoAttr`, `AnimationFunction`)

Die abgeleiteten Klassen von `AnimationFunction` (Abb. 27) bietet 10 Funktionen für die Modellierung von Aktionen. Die Modellierung der einzelnen Aktionen geschieht über Attribute von `ActionInfo` (Abb. 26). Für die 5 Animationsobjekte ist je ein Attribut für die Modellierung vorgesehen: `alpha`, `direction`, `scalefactor`, `shearfactor` und `tumblepath`. Diese 5 Attribute kennen folgende 5 Parameter:

- Zielwert der Transformation (fehlt bei `tumblepath`). Der Zielwert kann eine reelle Zahl, ein 2D-Vektor oder ein 3D-Vektor sein.

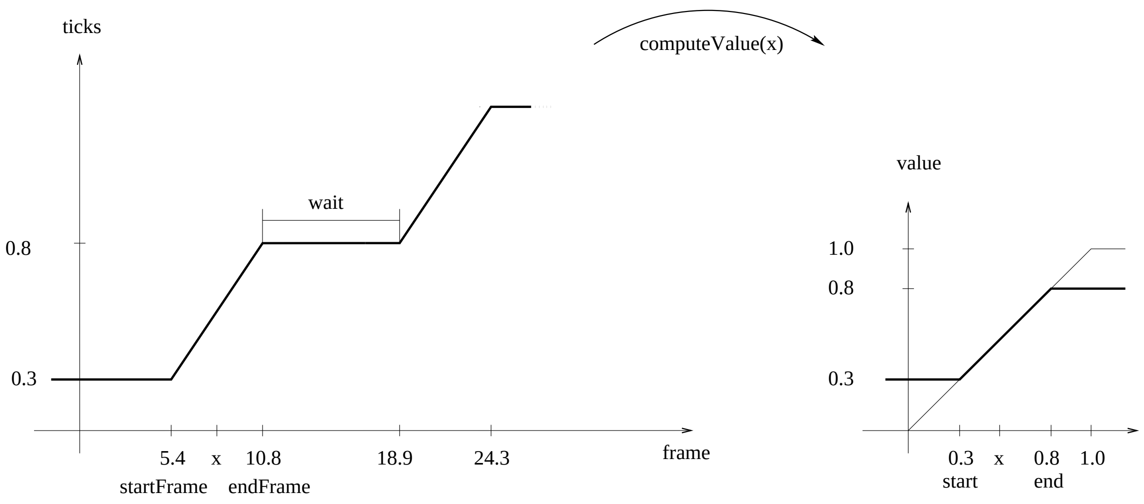


Abbildung 24: `computeTicks`.

- Funktionsname, Defaultwert `id`.
- Startwert (`start`) des Definitionsintervalles, Defaultwert `0`.
- Endwert (`end`) des Definitionsintervalles, Defaultwert `1`. Start- und Endwert bestimmen das Definitionsintervall der Modellierungsfunktion. Ausserhalb des Definitionsintervalles wird die Funktion entsprechend den Intervallgrenzwerten konstant fortgesetzt (siehe Formel (11)).
- Schrittweite (`step`), Defaultwert `0`. Die Schrittweite bestimmt die Stufenlänge der Treppenfunktion $step(x)$ (10). Als erstes wird die Treppenfunktion angewendet, allerdings nur, falls `step` > 0. Mit `step` > 0 wird der Verlauf der Transformation diskretisiert (Beispiel: Abb. 25).

Der Spezialfall `tumblepath` kennt keine direkte Angabe des Zielwertes (NURBS-Kurve), deshalb muss dieser wie ein Attribut nach den Parametern mit geschweiften Klammern angegeben werden. Diese Implementierungsweise erlaubt die volle Ausnutzung der NURBS-Library beim Einlesen einer Szenen. Die Parameter werden entsprechend der Reihenfolge der Liste eingegeben. Der Zielwert darf bei der Eingabe nicht fehlen. Bei der restlichen Eingabe der Parameter kann beliebig abgebrochen werden, wobei für fehlenden Parameter die Defaultwerte übernommen werden.

Die Treppenfunktion

$$step(x) = \begin{cases} step \cdot \lfloor \frac{x}{step} \rfloor & : \; step - \epsilon > 0 \\ x & : \; \text{sonst} \end{cases} \tag{10}$$

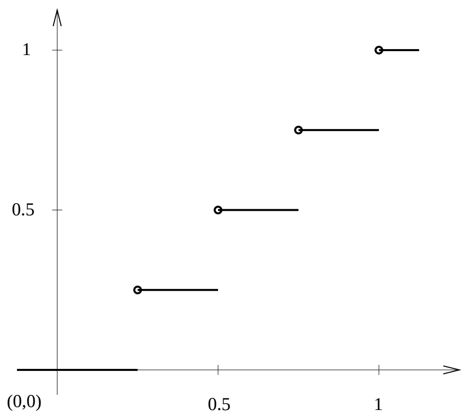


Abbildung 25: Beispiel von $step(x)$ für `step` $= \frac{1}{4}$.

Die Intervalltransformation

$$AFTransformation(x) = \begin{cases} start & : \ x \leq startFrame + \epsilon \\ end & : \ x \geq endFrame - \epsilon \\ start + \frac{step(x) - startFrame}{endFrame - startFrame}(end - start) & : \ sonst \end{cases} \quad (11)$$

Die Auswertung der Modellierungsfunktion

$$computeValue(x) = f(AFTransformation(x)) \quad (12)$$

$f(x)$ ist eine (Modellierungs-)Funktion aus der Tabelle (2) von Seite 42.

Um die Transformation korrekt definieren zu können, müssen je nach Transformationsart zusätzliche Angaben über die Transformation gemacht werden. Die Angaben werden mit den folgenden Attributen spezifiziert:

- `center`. Optionales Attribut für die Angabe des Zentrums für die Transformation `grow`, `shear` und `turn`. Falls das Attribut weggelasen wird, wird die Boundingbox als Zentrum angenommen.
- `axis`. Normalvektor oder Rotationsachse für `shear` bzw. `turn`.
- `tumblecenter`. Optionales Attribut für die Beschreibung des Rotationszentrums.
- `tumbledirection`. Optionales Attribut für die Beschreibung der Ausrichtung des Objektes. `tumblecenter` und `tumbledirection` bilden zusammen die Ausrichtung des Objektes.

Diese zusätzlichen Attribute kennen keine Modellierung mit Funktionen. Die Attribute erlauben aber eine genauere Spezifikation der Transformationen.

Klassendiagramme

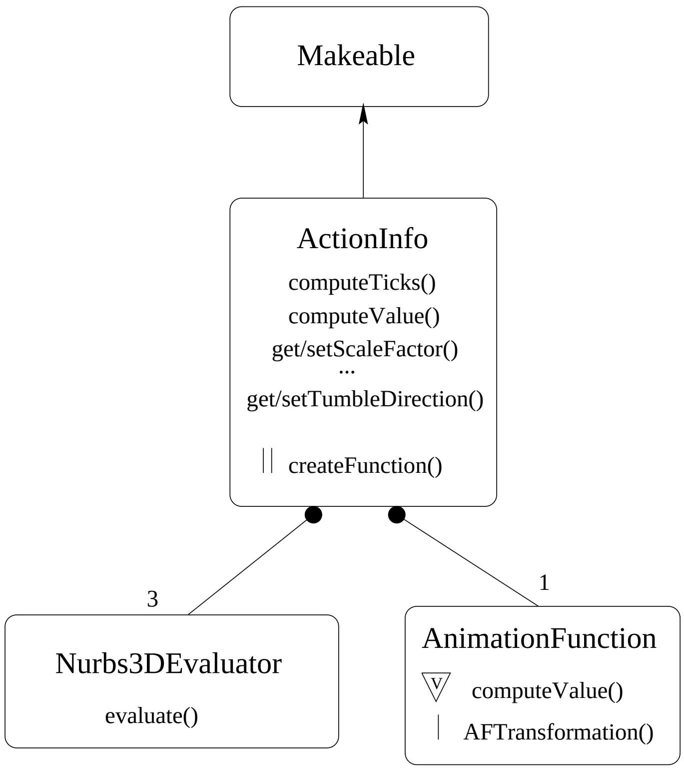


Abbildung 26: Klassendiagramm von `ActionInfo`.

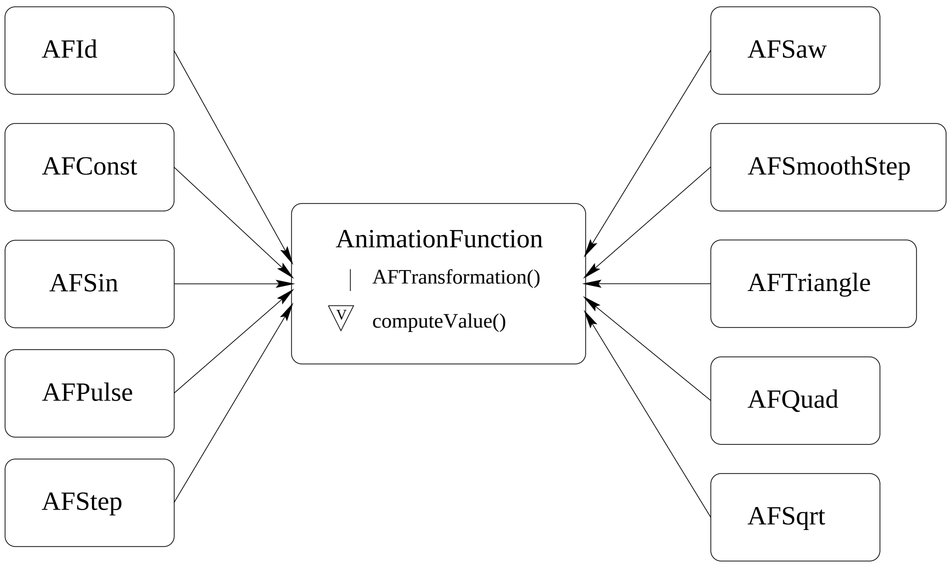


Abbildung 27: Klassendiagramm von `AnimationFunction`.

Die Modellierungsfunktionen im Einzelnen Auf dieser und auf den folgenden zwei Seiten sind alle Modellierungsfunktionen als Graph abgebildet und als mathematische Formel dargestellt. Alle Funktionen wurden mit `step` = 0 gezeichnet. Bei Unstetigkeitsstellen im Graph ist der Funktionswert mit einem kleinen Kreis dargestellt. Für das Studium der Modellierungsfunktionen sei auf `action.bsdl3` (Anhang D) verwiesen. Das Beispiel zeigt das zeitliche Verhalten der Funktionen (Anhang A).

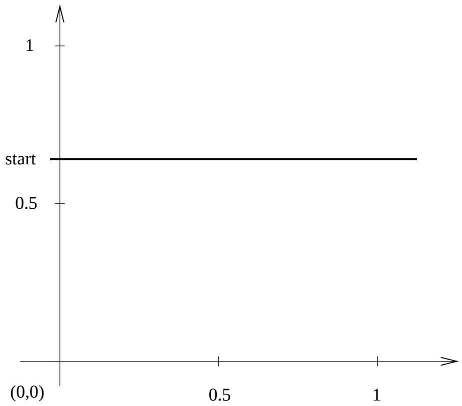


Abbildung 28: Konstante, `const`.

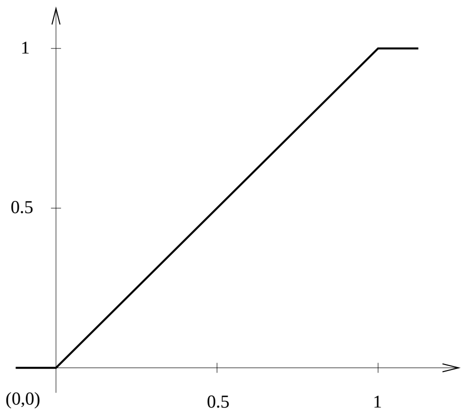


Abbildung 29: Identität, `id`.

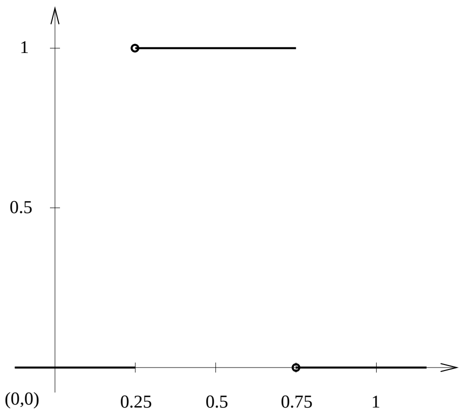


Abbildung 30: Pulsfunktion, `pulse`.

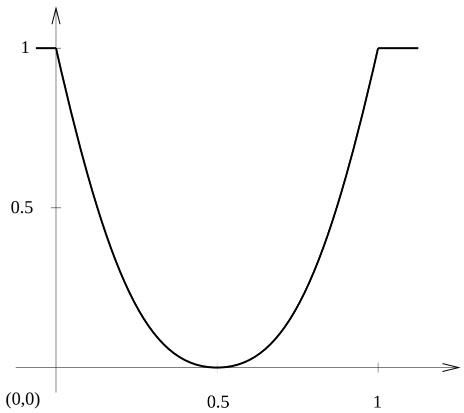


Abbildung 31: Quadratfunktion, `quad`.

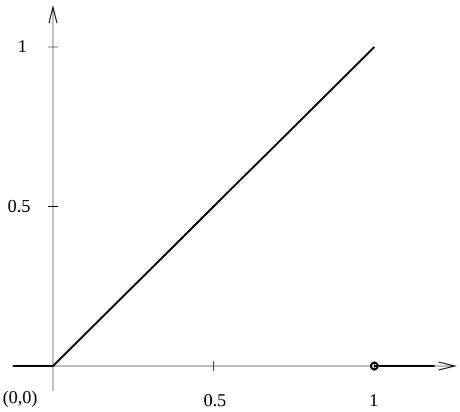


Abbildung 32: Sägezahnfunktion, `saw`.

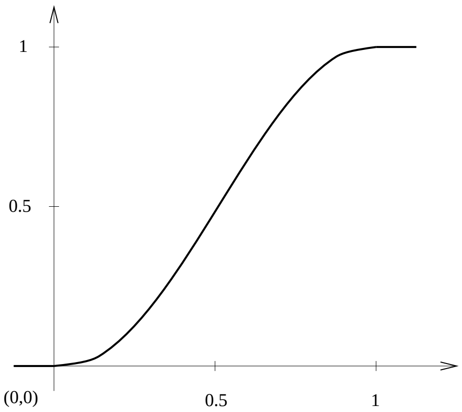


Abbildung 33: Geglättete Identität, `smoothstep`.

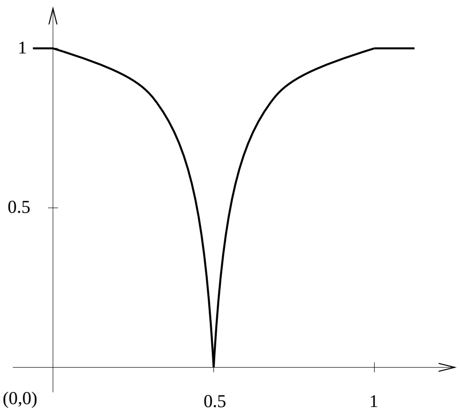


Abbildung 34: Wurzelfunktion, `sqrt`.

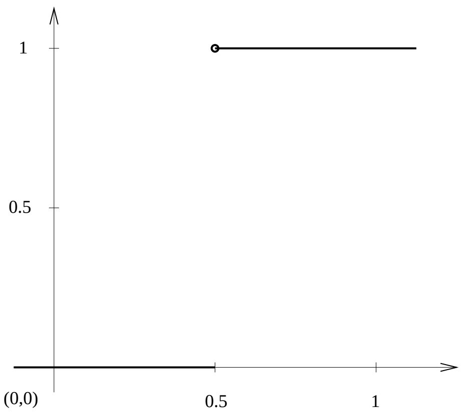


Abbildung 35: Treppenfunktion, `step`.

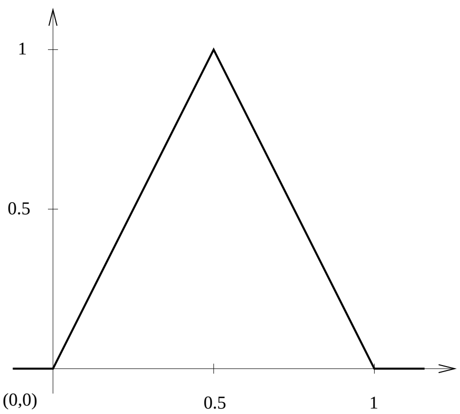


Abbildung 36: Dreiecksfunktion, `triangle`.

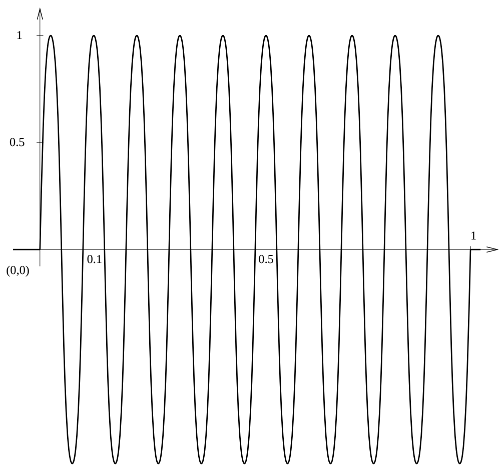


Abbildung 37: Sinusfunktion, `sin`.

Definition der Modellierungsfunktionen		
Funktion	Bezeichner	Definition
Konstante	`const`	$f(x) = start$
Identität	`id`	$f(x) = x$
Pulsfunktion	`pulse`	$f(x) = \begin{cases} 0 & : \; x < 0.25 - \epsilon \\ 0 & : \; x \geq 0.75 - \epsilon \\ 1 & : \; \text{sonst} \end{cases}$
Quadratfunktion	`quad`	$f(x) = (2(x - \frac{1}{2}))^2$
Sägezahnfunktion	`saw`	$f(x) = \begin{cases} 0 & : \; x \geq 1 - \epsilon \\ x & : \; \text{sonst} \end{cases}$
Geglättete Identität	`smoothstep`	$f(x) = x^2(3 - 2x)$
Wurzelfunktion	`sqrt`	$f(x) = \sqrt{\lvert 2x - 1 \rvert}$
Treppenfunktion	`step`	$f(x) = \begin{cases} 0 & : \; x < \frac{1}{2} - \epsilon \\ 1 & : \; \text{sonst} \end{cases}$
Dreiecksfunktion	`triangle`	$f(x) = \begin{cases} x < \frac{1}{2} & : \; 2x \\ x \geq \frac{1}{2} & : \; 2 - 2x \end{cases}$
Sinusfunktion	`sin`	$f(x) = sin(20\pi x)$

Tabelle 2: Definition der Modellierungsfunktionen.

3.8 Generieren von Animationen

Dieser Abschnitt befasst sich mit dem Generieren und dem Aufzeichnen von Animationen. Dies ist mit einer einfachen Applikation möglich, die folgende Funktionalitäten bietet:

- Die Szene einlesen oder erstellen.
- Aufsammeln aller Animationsobjekte.
- Grafische Darstellung der Szene.
- Alle Animationsobjekte mit dem entsprechenden frame-Wert ausführen.
- Darstellung speichern.

Das Erzeugen von Animationen bedingt als erstes eine Szene, die Animationsobjekte enthält, damit sich auch etwas bewegen lässt. Am einfachsten nimmt man eine schon modellierte Szene ohne Animationsobjekte und unterwirft die Objekte der Szene einer Transformation. Weiter muss eine geeignete Applikation die Szene einlesen und im Speicher ablegen. Es ist auch denkbar, dass die Szene direkt eingegeben (programmiert) oder über eine interaktive Oberfläche (3D-Editor) erstellt wird. Nun muss die ganze Szene dargestellt werden. Die Darstellungsart ist je nach Zweck verschieden. Bei einem Preview ist Wireframe sinnvoll, da oft nur ein grober Ablauf der Animation von Interesse ist. Für die Endfassung einer Animation ist eine aufwendige Raytracingmethode denkbar. Als nächstes muss an eine Speicherungsmöglichkeit der Bilder gedacht werden. Eine Möglichkeit ist die Speicherung von Einzelbildern als eine lose Folge. Nach Beenden der Animation werden dann die Bilder mit einem geeigneten Verfahren wie MPEG zu einer Filmsequenz zusammengefasst und komprimiert. Eine weitere Möglichkeit ist die direkte Erstellung der Filmsequenz mit der Applikation selbst. So fällt auch der Speicheraufwand für die Auslagerung der Einzelbilder weg. Diese Methode hat aber den Nachteil, dass Fehler nur durch eine Neuberechnung der ganzen Animation behoben werden können. Da die Berechnung einer Animation oft ungemein länger dauert als das Erstellen der Filmsequenz, ist bei anspruchsvolleren Animationen Vorsicht geboten. Der Besitz der Animation als lose Folge von Einzelbildern hat den Vorteil, dass gewisse Passagen neu berechnet werden können, ohne dass die ganze Animation neu berechnet werden muss. Als letzte Möglichkeit sei hier noch das direkte Abspielen auf Videoband oder das Belichten eines Filmes erwähnt. Diese Methoden haben den grossen Vorteil, dass sie ohne Komprimierungsverfahren wie MPEG auskommen.

3.8.1 Das Abspielen von Animationen

Für ein korrektes Abspielen der Animation sind folgende Punkte nötig.

- Alle Animationsobjekte aufsammeln und in einer Liste speichern.
- Die Szene in Ausgangsposition bringen.
- Die Animationsobjekte mit dem gewünschten frame-Werte ausführen.

Das Aufsammeln der Animationsobjekte Das Aufsammeln der Animationsobjekte dient der Vereinfachung, so dass man sich das Aufsammeln später beim Berechnen der einzelnen Frames ersparen kann. Beim Hinzufügen und beim Entfernen von Objekten müssen die Animationsobjekte neu aufgesammelt werden.

```
//
// Collect all animation objects in the world.
//
if (listAnimation != NULL)
  delete listAnimation;

listAnimation = new List<Animation3D*>;
Collector3DFor<Animation3D*> animCollector;
animCollector.apply(world);
for (animCollector.first(); !animCollector.isDone(); animCollector.next())
  listAnimation->append(animCollector.getObject());
```

Initialisierung der Animation Bevor eine Szene mit Animationsobjekten angezeigt wird, sollten die Animationsobjekte mit einem frame-Wert initialisiert werden, um die Szene in einen eindeutigen Zustand zu bringen.

```
//
//  Init Animation
//
for (long i=0; i<listAnimation->count(); i++)
  listAnimation->item(i)->frame(frame);
```

Abspielen der Animation Das Abspielen gestaltet sich einfach, indem die Funktion `doAnimation()` aufgerufen wird. `doAnimation()` führt die Animationsobjekte mit dem entsprechenden frame-Wert aus. Anhand des Rückgabewertes der Memberfunktion `frame` wird ein Redisplay ausgeführt.

```
void doAnimation()
{
   long count     = listAnimation->count();
   bool redisplay = false;
   frame         += frameStep;

   for (long i=0; i<count; i++)      // do animation
     if (listAnimation->item(i)->frame(frame)) redisplay = true;

   if (redisplay){                   // redisplay animation if needed
     world->getObjects()->computeBounds();
     glutPostRedisplay();

   }
}
```

3.8.2 Die Applikation `flythrough`

Für das Abspielen und Generieren von Bildfolgen habe ich keine eigene Applikation geschrieben, sondern habe `flythrough` erweitert. Die Applikation kann 3D Szenen einlesen und anschliessend visualisieren. Der Benutzer kann sich in der 3D Welt frei bewegen. Die Darstellung kann wahlweise als Drahtgittermodell oder als Flächenmodell in verschiedenen Qualitätsstufen schattiert erfolgen. Für das Generieren von MPEG-Dateien wurde die MPEGelib benutzt, die MPEG-Dateien direkt erzeugen kann. Im Menu kann das Speichern der Einzelbilder in eine MPEG-Datei ein- und ausgeschaltet werden. Zusätzlich kann noch das aktuelle Bild im BOOGA -Format pixi, PostScript oder PPM gespeichert werden. Um eine annehmbare interaktive Darstellung zu erzielen, bedient sich `flythrough` verschiedener 3D Beschleunigungstechniken. So werden Grundobjekte (Primitivelemente) wie Zylinder oder Kegel je nach sichtbarer Grösse in entsprechender Qualität dargestellt. Auf diese Weise werden kleine Objekte viel rascher dargestellt. Zusätzlich ist es möglich, die maximale Bildrate zu bestimmen. `flythrough` versucht entsprechend der zur Verfügung stehenden Zeit zwischen zwei Bildern so viel als möglich anzuzeigen. Diese Beschleunigungstechniken erlauben eine vernünftige interaktive Visualisierung ohne hardwarebeschleunigte Darstellung. Das freie Bewegen in der Szene ist auf zwei Arten möglich. Zum einen kann man sich im Walkthrough-Modus durch die 3D Szene bewegen, indem man sich entfernt, näher hingeht oder sich dreht. Im Inspect-Modus ist ein Bewegen in allen drei Dimensionen möglich. Ein vollständiger Überblick über die Menufunktionen gibt die Tabelle (4) auf Seite xi wieder. Die Tastenkommandos sind in der Tabelle (3) aufgelistet.

Tastenkommandos von `flythrough`	
Taste	Funktion
q, ESC	`flythrough` beenden
r	Szenenfile neu lesen
SPACE	Standardblickfeld
w	Szene als pixi speichern
W	Szene als PS speichern
s	Animation anhalten
i	Animation so schnell als möglich anzeigen
f	schrittweise vorwärts
b	schrittweise rückwärts
TAB	Ablaufrichtung ändern
c	Animation anhalten, Schrittweite 1 und Bild 0
a	Information über den aktuellen Stand der Animation
h, ?	Hilfetext

Tabelle 3: Tastenkommandos von `flythrough`.

Die MPEGelib und MPEG Eine schon erwähnte Möglichkeit der Komprimierung ist das Verfahren MPEG. Mit diesem Verfahren ist es möglich eine lose Folge von Einzelbildern zu einer Filmsequenz zusammenzufassen und den Speicheraufwand zu reduzieren. Das Verfahren benutzt die Kohärenz zwischen den Einzelbildern. Die Bilder sind allerdings mit kleinen Fehlern behaftet, die aber beim raschen Abspielen von blossem Auge kaum erkennbar sind. Das Erstellen von MPEG-Dateien mit dem `mpeg_encoder` von Berkley liefert sehr gute Ergebnisse. Dieses Programm kann aber nicht direkt in eine eigene Applikation eingebunden werden, da die Steuerung über ein Skript geschehen muss. Mit der Flut von Parametern wurden schon unzählige Skripts für die Steuerung des `mpeg_encoder` geschrieben. Hier greift die MPEGelib von Alex Knowles (`http://www.tardis.ed.ac.uk/~ark/mpegelib/`) nun ein. Die Library baut auf dem parallelen `mpeg_encoder` von Berkley auf. Diese Library kann in eigene Programme eingebunden werden und erlaubt das direkte Erstellen einer MPEG-Datei über ein paar einfache Libraryaufrufe:

- `MPEGe_open` initialisiert die Library und eröffnet eine MPEG-Datei.
- `MPEGe_image` für das Anfügen eines Einzelbildes an die MPEG-Datei.
- `MPEGe_close` schliesst die Library und die MPEG-Datei

Zusätzlich können beim Initialisieren der Library noch eigene Parameter für die Kodierung der MPEG-Datei angegeben werden.

3.9 Einschränkungen

Bei der Realisation der Animation musste ich ein paar Einschränkungen hinnehmen, die aber in den meisten Fällen elegant umgehbar sind. Die folgende Liste gibt einen Überblick über mögliche Problemfälle mit Animationsobjekten und deren Lösung.

- Ein Animationsobjekt überschriebt seine Transformationsmatrix mit seiner Animationsmatrix, d.h. alle lokalen Transformationen auf das Animationsobjekt sind nutzlos. Das Problem lässt sich einfach durch Abkapselung des Objektes mit einer Liste lösen.

- `tumble` kennt aus programmiertechnischen Gründen weder die Wiederholung noch die Pause für eine Aktion. Die Parameter `times` und `wait` werden ignoriert. Wiederholungen müssen deshalb explizit angegeben werden.

- `tumble` hat noch eine weitere Unannehmlichkeit, die bei der Benutzung der Ausrichtungsfunktionalität auftreten kann. `tumble` kann die Ausrichtung nicht garantieren, wenn der Ausrichtungsvektor entgegengesetzt zum Startvektor steht, weil die Rotationsachse nicht mehr eindeutig definiert ist. Wenn dieser Fall eintritt, geht `tumble` um ϵ auf der Kurve zurück und versucht es noch einmal. Bei einem weiteren Fehlschlag wird dann die Ausrichtung fallengelassen. Eine korrekte Ausrichtung wäre mit der etwas aufwendigeren Realisation mit einem Dreibein lösbar.

- Das Weglassen des optionalen Parameters `center` für die Transformationen `grow`, `shear` und `turn` kann fehlerhafte Bilder zur Folge haben, weil dann möglicherweise eine nicht aktuelle Boundingbox als Zentrum angenommen wird. Die Animation ist dann nicht mehr eindeutig definiert. Weil die Animationsobjekte in einer nicht vorhersehbaren Reihenfolge ausgeführt werden und die Szene auch während der Ausführung verändert werden kann, kann keine aktuelle Boundingbox des zu animierenden Objektes vorausgesetzt werden. Wenn möglich sollte man das Zentrum explizit mit `center` angeben, um solche Unannehmlichkeiten vorzubeugen. Es ist aber auch möglich dies zu umgehen, indem man die Ausführung aller Animationsobjekte entsprechend der maximalen Verschachtelungstiefe der Animationsobjekte nacheinander wiederholt, denn das Weglassen von `center` kann bei Animationsobjekten, die keine Animationsobjekte mehr enthalten und während der Ausführung unverändert bleiben, bedenkenlos angewendet werden.

- `shear` `tumble` und `turn` haben die unangenehme Eigenschaft, dass sie im allgemeinen nicht kommutieren. Bei Überlagerungen von Aktionen sei deshalb Vorsicht geboten, da sonst unerwünschte Transformationen entstehen. Es ist ratsam die Überlagerung zu verschachteln und jedem `shear` bzw. `turn` eine Aktion zuzuordnen. Für `tumble` sind Überlagerungen nicht ratsam.

- Die Animationsobjekte haben nicht zu jeder `set`-Methode eine entsprechende `get`-Methode.

- Beim Einfügen und Entfernen von Objekten müssen die Animationsobjekte neu aufgesammelt werden.

Damit ist das Kapitel über die Implementierung eines Ansatzes abgeschlossen. Das folgende Kapitel gibt eine praxisorientierte Einführung in die Computer-Animation für BOOGA anhand eines Beispieles.

4 Beispiel - Von der Idee zum MPEG-Film

Diese Kapitel gibt eine praxisorientierte Einführung in die Erstellung von Animationen mit BOOGA. An Hand eines Beispieles werde ich die nötigen Schritte von der Idee bis zum fertigen Film erklären. Das Erstellen einer Animation besteht aus den drei Hauptaktivitäten:

- Modellieren der Szene.
- Animieren der Objekte.
- Berechnung der Animation.

Das vollständige Beispiel ist im Anhang B auf Seite iii abgedruckt.

4.1 Die Idee

Als einfaches Beispiel wähle ich hier eine Kinderschaukel mit dem Ziel sie zu modellieren und anschliessend zu animieren. Die Schaukel besteht einerseits aus einem statischen Teil, dem Gerüst, und anderseits aus einem dynamischen Teil, der Schaukel selbst. Die animierte Schaukel soll am Ende eine ungebremste Pendelschwingung ausführen.

Abbildung 38: Die Kinderschaukel.

4.2 Das Modellieren der Szene

Die Modellierung der Kinderschaukel wurde vereinfacht, indem ich die Szene in Unterobjekte zerlegt habe. Das eine Unterobjekt enthält das Gerüst (statisch), das andere Unterobjekt enthält die Schaukel selbst, die später animiert werden soll. Zusätzlich ist noch ein Rasen definiert, damit die Schaukel nicht in der Luft hängt.

Definition des Rasens Der Rasen besteht aus einem simplen Polygon, das ein grünes Quadrat repräsentiert.

```
define ground list {
  // Definition des Rasens
  polygon ([-45,-45,0],[45,-45,0],[45,45,0],[-45,45,0]) { green; }
}
```

Definition des Gerüstes Das Gerüst lässt sich schon mit fünf Zylindern modellieren. Vier Zylinder dienen als Träger des fünften Zylinders. Die Schaukel wird später am Träger-Zylinder befestigt. Das Gerüst wird als Liste, bestehend aus Zylindern im Koordinatenursprung, aufgebaut. Die Zylinder sind grau und im Verhältnis zu ihrer Länge dünn.

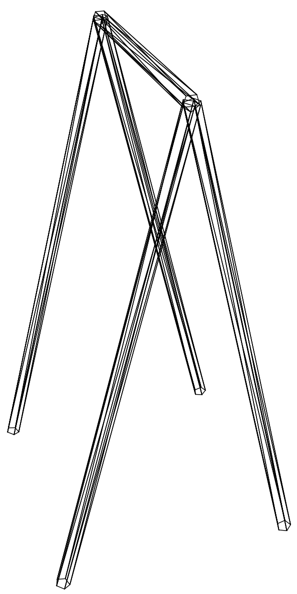


Abbildung 39: Das Gerüst der Kinderschaukel.

```
define stage list {
  cylinder (.3, [5,5,0],[5,0,20])   {grey;}  // Definition
  cylinder (.3, [5,-5,0],[5,0,20])  {grey;}  // des Geruestes
  cylinder (.3, [-5,5,0],[-5,0,20]) {grey;}
  cylinder (.3, [-5,-5,0],[-5,0,20]){grey;}
  cylinder (.3, [-5,0,20],[5,0,20]) {grey;}
}
```

Definition des Schaukelsitzes mit Aufhängung Die Schaukel besteht aus einem Brett, hier mit einer braunen Box modelliert, und der Aufhängung, die mit zwei sehr dünnen Zylindern dargestellt wird. Die Schaukel befindet sich vorerst in der Ruhelage. Die Lage bestimmt später das Modellieren der Pendelbewegung, deshalb habe ich hier eine möglichst einfache Lage gewählt.

```
define seat list {
  cylinder (.1, [3,0,20],[3,0,5])   { grey;}  // Definition des
  cylinder (.1, [-3,0,20],[-3,0,5]) { grey;}  // Sitzes
  box ([-3.5,-1,4.8],[3.5,1,5])     { brown;}
}
```

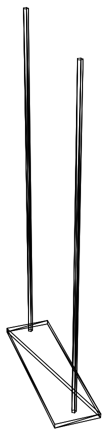


Abbildung 40: Der Schaukelsitz mit Aufhängung.

Definition der Kamera, der Lichtquelle und der Farben

```
camera {                 // Definition der Kamera
  perspective {
    eye [30,-10,30];        // Kamerastandort
    lookat [0, 0, 10];      //
    resolution (512, 512);  // Bildaufloesung
  }
  background [.3,.3,.3];
};

pointLight (2, [1,1,1]) { position [  500,   500, 500]; } // Die Lichtquelle

define green   matte { diffuse [.2,.5,.2]; }
define brown   matte { diffuse [.3,.2,.2]; }
define grey    matte { diffuse [.7,.7,.7]; }
```

4.3 Animieren der Objekte

Die Kinderschaukel wird nun aus Gerüst und Schaukel zusammengesetzt. Das Gerüst ist statisch und kann so übernommen werden. Die Schaukel selbst muss nun mit einer Pendelschwingung in Bewegung gebracht werden. Als erstes muss eine Transformation gefunden werden, welche die Pendelbewegung wiedergeben könnte. Naheliegend ist die Rotation, die hier als Drehung um den Träger-Zylinder verstanden wird. Demzufolge ist dieser Zylinder das Rotationszentrum und die Rotationsachse entspricht der Ausrichtung des Zylinders. Nun muss eine geeignete Modellierung der Rotation gefunden werden. Das Schwingen lässt sich durch die Differentialgleichung (13) beschreiben, die aber ohne die Approximation (15) analytisch nicht lösbar ist.

$$\frac{d^2\alpha}{dt^2} = -c \cdot \sin\alpha \tag{13}$$

$$c = \frac{g}{l} \tag{14}$$

Die Lösung (16) ist für kleine α eine parktikable Lösung der Differentialgleichung. Daraus folgt, dass die Schaukel mit der Sinusfunktion (siehe Seite 37) modelliert werden muss.

$$\alpha \approx \sin\alpha \tag{15}$$

$$\alpha(t) = \alpha_0 \cdot \sin(t\sqrt{c}) \tag{16}$$

Eine ganze Pendelbewegung (einen Zyklus) setzt sich nun aus zwei Aktionen zusammen. Die Schaukel hebt sich mit einer Aktion aus der Ruhelage auf die einen Seite und fällt wieder. Die andere Seite wird bis auf den negativen Drehwinkel identisch modelliert. Nun müssen noch diese zwei Aktionen untereinander abgestimmt werden. Die erste Aktion ist von Frame 0 bis 12 und die zweite von Frame 12 bis 24 definiert. Mit der Option `wait` 12 werden beide aufeinander abgestimmt, so dass genau nur eine Aktion aktiv ist. Die Option `times` 0 lässt die Aktionen endloss ausführen.

Definition der Schaukel

```
define swing list {  // Definition der Schaukel
  stage;                    // Das Geruest
  turn {                    // Rotation einleiten
    action (0,12,0,12) {       // Rotation von Frame 0 bis 12, 12 Frames warten,
                                 // unendlich wiederholen
      center ([0,0,20]);         // Rotationszentrum (0,0,20)
                                 //        (5. Zylinder des Geruestes)
      axis ([1,0,0]);            // Rotationsachse (1,0,0)
      alpha (60, "sin",0,0.05,0);// Drehung um 60 Grad
                                 // mit Sinusfunktion modelliert
                                 // auf dem Intervall [0,0.05] definiert
                                 // mit Schrittweite 0, d.h kontinuierlich
                                 // entspricht dem Schwingen hinten
    }
    action (12,24,0,12) {      // Rotation von Frame 12 bis 24, 12 Frames warten,
                                 // unendlich wiederholen
      center ([0,0,20]);
      axis ([1,0,0]);
      alpha (-60, "sin",0,0.05,0);
                                 // entspricht dem Schwingen vorne
    }
    seat;                  // Der Schaukelsitz mit Aufhaengung
  }
}
```

Die Pendelbewegung habe ich in einer ersten Variante mit zwei Aktionen beschrieben um die Modellierung mit mehreren Aktionen zu zeigen. Es ist natürlich möglich die Bewegung mit einer einzigen Aktion zu beschrieben. In einer zweiten Variante nütze ich die spezielle Definition der Sinusfunktion (siehe Seite 41) aus, die als einzige Funktion negative Funktionswerte kennt. Die zwei Aktionen können nämlich äquivalent in einer Aktion wie folgt zusammengefasst werden:

```
    action (0,24,0) {  // Rotation von Frame 0 bis 24, unendlich wiederholen
                       // gleichwertige Loesung wie mit zwei Aktionen
      center ([0,0,20]);
      axis ([1,0,0]);
      alpha (60, "sin",0,0.1,0);
    }
```

Sind einmal die Transformationen modelliert und die Objekte definiert, so muss die ganze Szene mit Instanzen aufgebaut werden, damit man auch etwas sehen kann. Die Szene (siehe Abb. 38 auf Seite 48) besteht nun aus den zwei Instanzen `ground` und `swing`.

```
ground;
swing;
```

Der Vorteil mit `define` zu arbeiten ist, dass ein Objekt beliebig oft wiederverwendet werden kann. In diesem Beispiel könnte die Kinderschaukel mehrmals aufgeführt und mit einer Translation positioniert werden. Damit stände der Modellierung eines grösseren Kinderspielplaztes nichts im Wege, ausser dass die Schaukeln synchron pendeln würden.

4.4 Abspielen und Aufzeichnen der Animation

Für das Abspielen und Aufzeichen kann die Applikation `flythrough` benutzt werden, die einerseits die Animation anzeigen kann und anderseits das Herumlaufen in der Szene bietet. Beim Aufruf von `flythrough` kann optional der Name (ohne Endungen) für Ausgabendateien angegeben werden. Nachdem die Schaukel (`swing.bsdl3`) eingelesen wurde, kann sie in die gewünschte Lage gedreht und eine passende Darstellung gewählt werden. Mit der Taste `i` wird die Animation gestartet und mit `s` wieder angehalten. Für das Erstellen eines MPEG-Filmes sollte eine Bildschirmauflösung von etwa 150×150 gewählt werden. Mit `c` wird die Animation wieder in die Ausgangslage (Frame $= 0$ und Framerate $= 1$) gebracht (weitere Interaktionen sind in der Tabelle 3 auf Seite 45 aufgeführt). Für die Erstellung des MPEG-Filmes nutzte ich die zyklische Bewegung der Schaukel aus, indem ich einen einzigen Zyklus in eine MPEG-Datei speichere und diesen mit einem MPEG-Player wiederhole. Diese Technik des Wiederholens eines Zyklus wurde schon in Kapitel 2 unter *cycle animation* erläutert. Ein geeigneter Zyklus ist derjenige von Frame 0 bis 24 (siehe Definition der Schaukel). Die Framerate wird gleich $\frac{1}{2}$ gesetzt, d.h. der Zyklus besteht aus genau 48 Einzelbildern. Im Menu muss unter `Animation` der Punkt `Save Frames : No` angewählt werden, um das automatische Speichern einzuschalten. Nun kann die Animation gestartet werden. Die Bildnummer wird nach erfolgtem Speichern eines Einzelbildes ausgegeben. Die Animation wird nach dem 47. Bild angehalten, weil nur 48 Bilder gespeichert werden müssen. Die MPEG-Datei wird durch wiederholtes Anwählen des Punktes `Save Frames : Yes` korrekt geschlossen. Während des Speicherns kann der Ablauf mit den Funktionen `s`, `f` oder `b` manipuliert werden, wobei man darauf achten sollte `flythrough` nicht herumzuschieben und die Bildauflösung zu belassen. `flythrough` ist in diesem Punkt noch verbesserungsfähig.

5 Schlussgedanken

Mit der vorliegenden Arbeit habe ich gezeigt, dass mit relativ kleinem Aufwand eine Animationssprache mit einem entsprechenden Animationsansatz entwickelbar ist. Die Ergebnisse lassen den Schluss zu, dass die algorithmische Animation einen brauchbaren Ansatz liefert.

Im Verlauf des Projektes traten einige naheliegende Erweiterungen hervor, aber auch Mängel oder Probleme mit BOOGA tauchten auf. Die folgende Liste wiedergibt mögliche Erweiterung wie auch Mängel:

- Beispiele, die nicht dokumentiert sind, sind unbrauchbar.
- Kameras und Lichtquellen sollten wie 3D-Objekte transformiert werden können.
- NURBS-Editor.
- Optionales Attribut, das nicht nur Bildauflösung sondern auch die Darstellung der Szene bestimmen kann.
- Eine Methode, die anhand der Komplexität einer Szene und den vorhandenen Computerressourcen einen möglichen Vorschlag für die Darstellungsart und die Bildauflösung liefert.
- Das Zurückschreiben einer Szenen. Diese Option ist für ein interaktives Editieren unerlässlich.
- Eine zusätzliche Modellierungsfunktion, die als 2D-NURBS definiert wird.
- Für die Berechnung der Endfassung einer Animation ist eine Applikation nötig, die nur Animationen berechnen und speichern kann und ohne Bildausgabe auskommt.
- Weitere Animationsobjekte, die 3D-Objekte deformieren können. Ich denke an dynamische Animation.
- Optionale Attribute, die Beginn, Ende und Schrittweite einer Animation definieren.
- Attribute, die Animationsobjekte zeitlich skalieren und verschieben können. Damit könnte man jede einzelne Instanz eines animierten Objektes zeitlich kontrollieren.
- Verschachtelte Bewegungsabläufe, wie das Gehen eines Menschen, sind von Hand nur schwer zu realisieren. Hier sind Tools nötig, die aus Messdaten den Bewegungsablauf in BSDL wiedergeben können.
- Eine Schicht auf den Animationsobjekten aufbauend, die komplexe Bewegungen und Deformationen versteht.
- Kameras, Lichtquellen und alle 3D-Objekte als 4D-Objekte definieren und Animationen durch die Verschiebung der Hyperebene im 4D-Objektraum erzeugen. Dieser Ansatz ist mit dem Ansatz dieser Arbeit unverträglich.

Literatur

[Abdel93] R. Abdelhamid. *Das Vieweg LaTeX–Buch: eine praxisorientierte Einführung.* Vieweg, 2. verb. Auflage, 1993.

[Amann91] S. Amann. *Modellierung von Bewegung.* IAM Universität Bern, 1991.

[Bächler95] R. Bächler. *Entwurf und Implemtierung einer NURBS-Library.* IAM Universität Bern, 1995.

[Bieri94] H. Bieri. *Vorlesung 3D-Grafik.* IAM Universität Bern, 1994.

[Bieri95] H. Bieri. *Vorlesung Geometrisches Modellieren.* IAM Universität Bern, 1995.

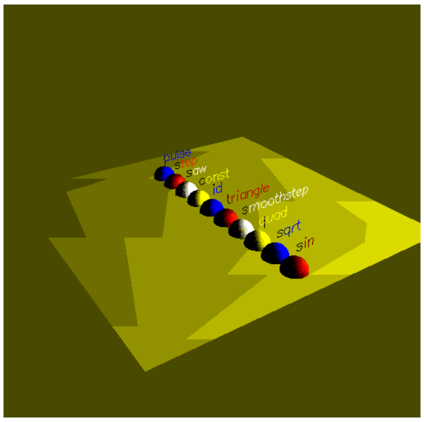
[FvDFH90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics, Principles and Practice.* Addison Wensley, second edition, 1990.

[Ebert94] D.F. Ebert, F.K. Musgrave, D. Peachey, K. Perlin, S. Worley. *Texturing and Modeling, A Procedural Approach.* Academic Press, 1994.

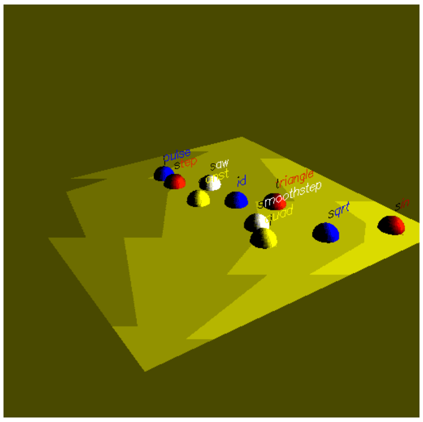
[MTT90] N. Magnenat-Thalmann and D. Thalmann. *Computer Animation Theory and Practice.* Springer Verlag, 1991.

[Wern94] J. Wernecke. *The Inventor Mentor.* Addison Wensley, second edition, 1994.

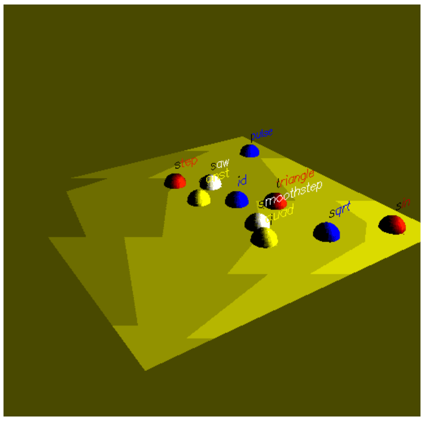
A Galerie



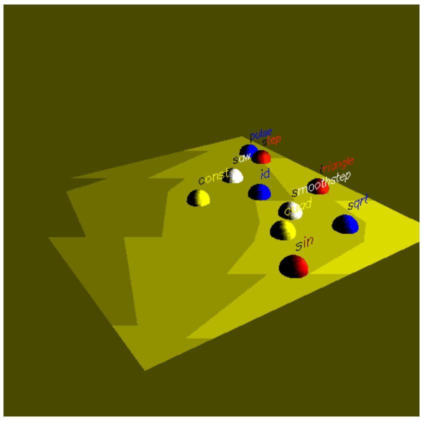
`action.bsdl3`, frame = 0.



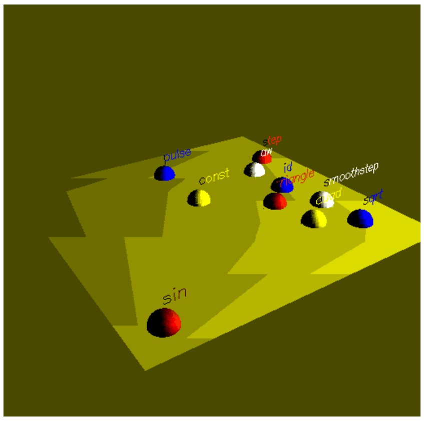
`action.bsdl3`, frame = 24.9.



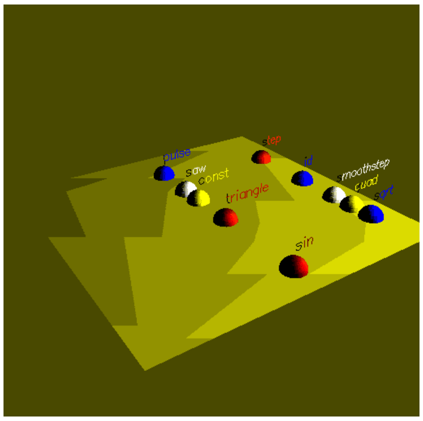
`action.bsdl3`, frame = 25.



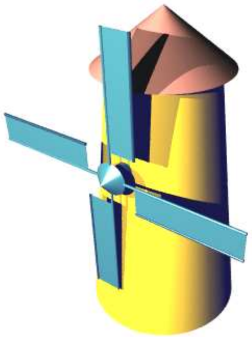
`action.bsdl3`, frame = 50.



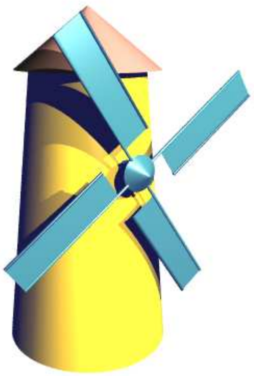
`action.bsdl3`, frame = 75.



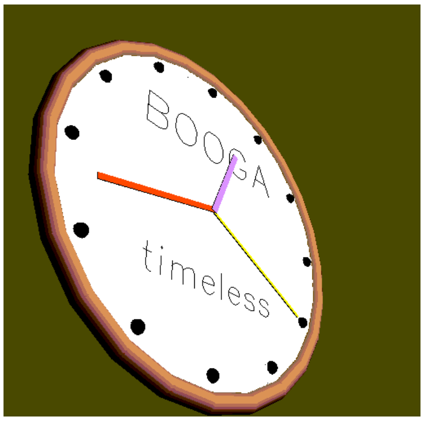
`action.bsdl3`, frame = 100.



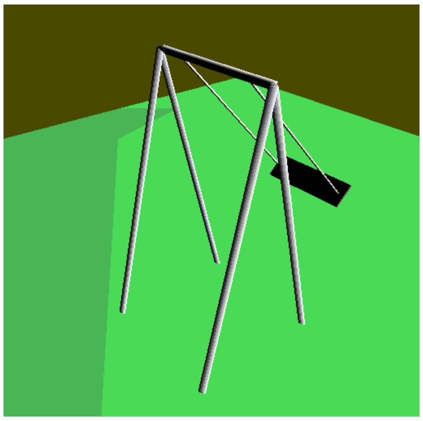
`windmill.bsdl3`, Rayshade.



`windmill.bsdl3`, Rayshade.



`myclock.bsdl3`, frame = 2600.



`swing.bsdl3`, frame = 3.

B Kinderschaukel

```
/*
 * swing.bsdl3
 *
 * Copyright (C) 1996, Thierry Matthey <matthey@iam.unibe.ch>
 *                     University of Berne, Switzerland
 *
 * All rights reserved.
 *
 * This software may be freely copied, modified, and redistributed
 * provided that this copyright notice is preserved on all copies.
 *
 * You may not distribute this software, in whole or in part, as part of
 * any commercial product without the express consent of the authors.
 *
 * There is no warranty or other guarantee of fitness of this software
 * for any purpose.  It is provided solely "as is".
 *
 */

using 3D;

//
// Deinition der Kamera und Lichtquellen
//
camera {                 // Definition der Kamera
  perspective {
    eye [30,-10,30];        // Kamerastandort
    lookat [0, 0, 10];      //
    resolution (512, 512);  // Bildaufloesung
  }
  background [.3,.3,.3];
};

pointLight (2, [1,1,1]) { position [  500,   500, 500]; } // Die Lichtquelle

//
// Deinition Farben
//
define green   matte { diffuse [.2,.5,.2]; }
define brown   matte { diffuse [.3,.2,.2]; }
define grey    matte { diffuse [.7,.7,.7]; }

//
// Definition Hintergrund
//
define ground list {
  // Definition des Rasens
  polygon ([-45,-45,0],[45,-45,0],[45,45,0],[-45,45,0]) { green; }
}
```

```
//
// Definition Geruest
//
define stage list {
  cylinder (.3, [5,5,0],[5,0,20])    {grey;} // Definition
  cylinder (.3, [5,-5,0],[5,0,20])   {grey;} // des Geruestes
  cylinder (.3, [-5,5,0],[-5,0,20])  {grey;}
  cylinder (.3, [-5,-5,0],[-5,0,20]) {grey;}
  cylinder (.3, [-5,0,20],[5,0,20])  {grey;}
}

//
// Definition Schaukelsitz
//
define seat list {
  cylinder (.1, [3,0,20],[3,0,5])   { grey;}  // Definition des
  cylinder (.1, [-3,0,20],[-3,0,5]) { grey;}  // Sitzes
  box ([-3.5,-1,4.8],[3.5,1,5])     { brown;}
}

//
// Deinition Schaukel
//
define swing list {
  stage;                    // Das Geruest
  turn {                    // Rotation einleiten
    action (0,12,0,12) {       // Rotation von Frame 0 bis 12, 12 Frames warten,
                                 // unendlich wiederholen
      center ([0,0,20]);         // Rotationszentrum (0,0,20)
                                 //       (5. Zylinder des Geruestes)
      axis ([1,0,0]);            // Rotationsachse (1,0,0)
      alpha (60, "sin",0,0.05,0);// Drehung um 60 Grad
                                 // mit Sinusfunktion modelliert
                                 // auf dem Intervall [0,0.05] definiert
                                 // mit Schrittweite 0, d.h kontinuierlich
                                 // entspricht dem Schwingen hinten
    }
    action (12,24,0,12) {      // Rotation von Frame 12 bis 24, 12 Frames warten,
                                 // unendlich wiederholen
      center ([0,0,20]);
      axis ([1,0,0]);
      alpha (-60, "sin",0,0.05,0);
                                 // entspricht dem Schwingen vorne
    }
    seat;                  // Der Schaukelsitz mit Aufhaengung
  }
}

// ---------------------------------------------

ground;
swing;
```

C Windmühle

```
/*
 * windmill.bsdl3
 *
 * Copyright (C) 1996, Bernhard Buehlmann <buehlmann@iam.unibe.ch>
 *                     Thierry Matthey <matthey@iam.unibe.ch>
 *                     University of Berne, Switzerland
 *
 * All rights reserved.
 *
 * This software may be freely copied, modified, and redistributed
 * provided that this copyright notice is preserved on all copies.
 *
 * You may not distribute this software, in whole or in part, as part of
 * any commercial product without the express consent of the authors.
 *
 * There is no warranty or other guarantee of fitness of this software
 * for any purpose.  It is provided solely "as is".
 *
 */

using 3D;

camera {
  perspective {
    eye [0,-20,5];
    lookat [0,0,5];
    eyesep 30;
  }
  background [.5,.5,.5];

}

pointLight (1, [1,1,1]) {
  position [4,20,17];
}

pointLight (1, [1,1,1]) {
  position [-4,20,17];
}

define green phong {
  ambient [.1,.1,.3];
  diffuse [42/255,94/255,73/255];
  specular [.9,.9,.9];
  specpow 20;
}
define darkGreen phong {
  ambient [.1,.1,.3];
  diffuse [42/455,94/455,73/455];
  specular [.9,.9,.9]; specpow 20;
```

```
}
define white phong {
  ambient [.1,.1,.3];
  diffuse [242/255,194/255,0/255];
  specular [.9,.9,.9];
  specpow 20;
}

define dachRot whitted {ambient [.1,.1,.1]; diffuse [190/255,115/255,97/255];}

define fluegel list {
  cylinder (.05, [0,0,0], [0,0,5]) {green;}
  list {
    box ([0,0,1], [-1,.01,5]) {green;}
    cylinder (.05, [-1,0,1], [-1,0,5]) {darkGreen;}
    rotateZ (-10);
  }
}

define rotor list {
  cone (.01, [0,.5,0], .6, [0,0,0]) {green;}
  cylinder (.3, [0,0,0], [0,-1,0]) {green;}

  fluegel { rotateY (90);}
  fluegel { rotateY (-90);}
  fluegel { rotateY (180);}
  fluegel { rotateY (0);}
}

define windmill list {
  cone (3, [0,0,0], 2, [0,0,10]) {white;}
  cone (2.5, [0,0,10], .01, [0,0,12]) {dachRot;}

  turn {                              // Rotation of secondhand
    action (0,60,0) {                 // from frame 0 to 60, endless
      center [0,2,7];                 // Rotationcenter (0,0,0)
      axis [0,1,0];                   // Rotationaxis (1,0,0)
      alpha (-360,"id", 0, 1, 1/60); // Constant rotation of -360 degres
                                      //  in 60 frames with a step of -6 degrees
    }
    rotor {translate [0,3,7];}
  }
}

//____________________________________________________

windmill;
```

D Modellierungsfunktionen

```
/*
 * action.bsdl3
 *
 * An example of all 10 functions to control the motion of a
 * transformation. The balls are moving with the motion
 * defined by the functions.
 * The animation is from frame 0 to 100.
 *
 * Copyright (C) 1996, Thierry Matthey <matthey@iam.unibe.ch>
 *                     University of Berne, Switzerland
 *
 * All rights reserved.
 *
 * This software may be freely copied, modified, and redistributed
 * provided that this copyright notice is preserved on all copies.
 *
 * You may not distribute this software, in whole or in part, as part of
 * any commercial product without the express consent of the authors.
 *
 * There is no warranty or other guarantee of fitness of this software
 * for any purpose.  It is provided solely "as is".
 *
 * -----------------------------------------------------------------------------
 *  $Id: action.bsdl3,v 1.2 1996/04/12 07:37:15 streit Exp $
 * -----------------------------------------------------------------------------
 */

using 3D;

//
// Definition of the camera and the lightsource
//

camera {
  perspective {
    eye [60,-40,40];
    lookat [0, 0, 0];
    resolution (512, 512);
  }
  background [.3,.3,.3];
};

pointLight (2, [1,1,1]) { position [  40,   40, 20]; }

//
// Definition of colors
//

define blue  matte { diffuse [0,0,1]; }
define red   phong { diffuse [.4,.1,.1]; specular [.6,0,0]; specpow 5;}
```

```
define yellow  matte { diffuse [1,1,0]; }
define white  matte { diffuse [1,1,1]; }

//
// Definition of the 10 animated balls with a different motion.
//

define balls list{
  move {                                  // translation
    action (0,100) {                      // from frame 0 to 100
      direction ([0,20,0],"pulse",0,1,0); // translation of (0,20,0) with the
                                          //  function "pulse" defined on
    }                                     //  [0,1] in 100 frames and no step
    list {                                // the ball with the text "pulse"
      sphere (2, [-16, 0, 0]) { blue;};
      text (2, 0.1, "pulse") {
        font "ROMAN";
        blue;
        translate [-16,0,3];
      }
    }
  }
  move {                                  // translation
    action (0,100) {                      // from frame 0 to 100
      direction ([0,20,0],"step",0,1,0);  // translation of (0,20,0) with the
                                          //  function "step" defined on
    }                                     //  [0,1] in 100 frames and no step
    list {                                // the ball with the text "step"
      sphere (2, [-12, 0, 0]) { red;};
      text (2, 0.1, "step") {
        font "ROMAN";
        red;
        translate [-12,0,3];
      }
    }
  }
  move {                                 // translation
    action (0,100) {                     // from frame 0 to 100
      direction ([0,20,0],"saw",0,1,0);  // translation of (0,20,0) with the
                                         //  function "saw" defined on
    }                                    //  [0,1] in 100 frames and no step
    list {                               // the ball with the text "saw"
      sphere (2, [-8, 0, 0]) { white;};
      text (2, 0.1, "saw") {
        font "ROMAN";
        white;
        translate [-8,0,3];
      }
    }
  }
  move {                                   // translation
    action (0,100) {                       // from frame 0 to 100
```

```
      direction ([0,20,0],"const",0,1,0);  // translation of (0,20,0) with the
                                           //  function "const" defined on
    }                                      //  [0,1] in 100 frames and no step
    list {                                 // the ball with the text "const"
      sphere (2, [-4, 0, 0]) { yellow;};
      text (2, 0.1, "const") {
        font "ROMAN";
        yellow;
        translate [-4,0,3];
      }
    }
  }
  move {                                 // translation
    action (0,100) {                     // from frame 0 to 100
      direction ([0,20,0],"id",0,1,0);   // translation of (0,20,0) with the
                                         //  function "id" defined on
    }                                    //  [0,1] in 100 frames and no step
    list {                               // the ball with the text "id"
      sphere (2, [0, 0, 0]) { blue;};
      text (2, 0.1, "id") {
        font "ROMAN";
        blue;
        translate [0,0,3];
      }
    }
  }
  move {                                    // translation
    action (0,100) {                        // from frame 0 to 100
      direction ([0,20,0],"triangle",0,1,0);// translation of (0,20,0) with the
                                            //  function "triangle" defined on
    }                                       //  [0,1] in 100 frames and no step
    list {                                  // the ball with the text "triangle"
      sphere (2, [4, 0, 0]) { red;};
      text (2, 0.1, "triangle") {
        font "ROMAN";
        red;
        translate [4,0,3];
      }
    }
  }
  move {                                      // translation
    action (0,100) {                          // from frame 0 to 100
      direction ([0,20,0],"smoothstep",0,1,0);// translation of (0,20,0) with
                                              //  the function "smoothstep"
                                              //  defined on [0,1] in 100
    }                                         //  frames and no step
    list {                                    // the ball with the text "smoothstep"
      sphere (2, [8, 0, 0]) { white;};
      text (2, 0.1, "smoothstep") {
        font "ROMAN";
        white;
        translate [8,0,3];
```

```
      }
    }
  }
  move {                                // translation
    action (0,100) {                    // from frame 0 to 100
      direction ([0,20,0],"quad",0.5,1,0);// translation of (0,20,0) with the
                                        //  function "quad" defined on
    }                                   //  [0.5,1] in 100 frames and no step
    list {                              // the ball with the text "quad"
      sphere (2, [12, 0, 0]) { yellow;};
      text (2, 0.1, "quad") {
        font "ROMAN";
        yellow;
        translate [12,0,3];
      }
    }
  }
    move {                              // translation
    action (0,100) {                    // from frame 0 to 100
      direction ([0,20,0],"sqrt",0.5,1,0);// translation of (0,20,0) with the
                                        //  function "sqrt" defined on
    }                                   //  [0.5,1] in 100 frames and no step
    list {                              // the ball with the text "sqrt"
      sphere (2, [16, 0, 0]) { blue;};
      text (2, 0.1, "sqrt") {
        font "ROMAN";
        blue;
        translate [16,0,3];
      }
    }
  }
  move {                                // translation
    action (0,100) {                    // from frame 0 to 100
      direction ([0,20,0],"sin",0,0.1,0); // translation of (0,20,0) with the
                                        //  function "sin" defined on
    }                                   //  [0,0.1] in 100 frames and no step
    list {                              // the ball with the text "sin"
      sphere (2, [20, 0, 0]) { red;};
      text (2, 0.1, "sin") {
        font "ROMAN";
        red;
        translate [20,0,3];
      }
    }
  }
}

//_______________________________________________________________

polygon ([-25,-25,0],[25,-25,0],[25,25,0],[-25,25,0]) { yellow; }
balls;
```

E Menufunktionen von `flythrough`

Menufunktionen von `flythrough`	
Application mode	
	Walkthrough
	Inspect
	Pick
	Raypaint
Rendering quality	
	Solid Gouraud (best quality, slowest)
	Solid Flat
	Wireframe Gouraud
	Wireframe Flat
	Wireframe
	Bounding Box (fastest)
Motion quality	
	Solid Gouraud (best quality, slowest)
	Solid Flat
	Wireframe Gouraud
	Wireframe Flat
	Wireframe
	Bounding Box (fastest)
Tools	
	Save framebuffer as pixi
	Save framebuffer as PS
	Save framebuffer as PPM
Options	
	Frames/sec
	Time spent in backbuffer
	Toggle statistics
	Toggle culling back faces
Animation	
	Save frames Yes/No
	Animation Off
	Frames/sec
	Framestep
Quit	

Tabelle 4: Menufunktionen von `flythrough`.