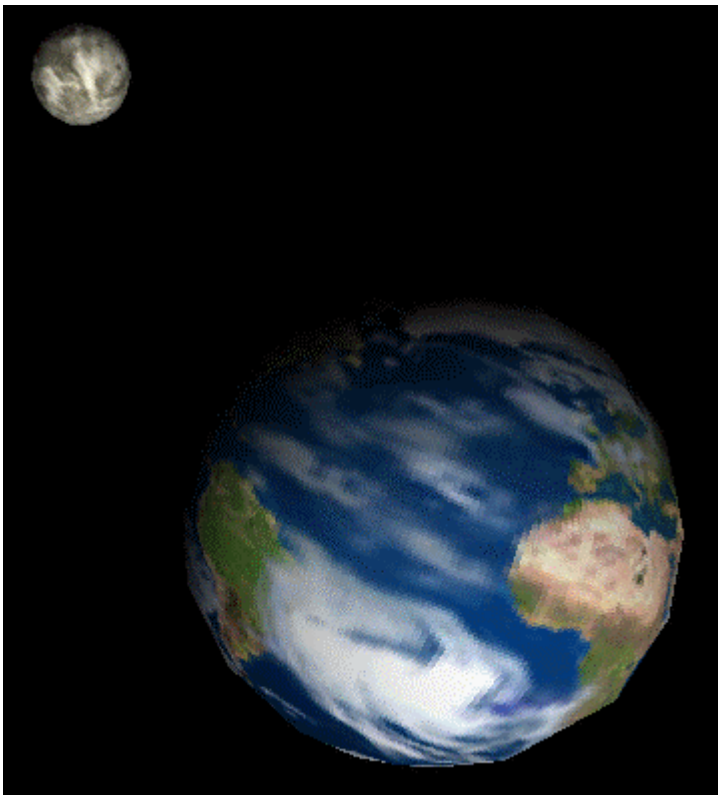
Multimedia-Labor

WS1999/2000



Andreas Junghans, Jürgen Baier

Inhaltsverzeichnis

1. Die Benutzerschnittstelle 3
2. Erstellung des VRML-Modells 6
   2.1. Das Koordinatensystem 6
   2.2. Die Planeten 6
      2.2.1. Form, Oberfläche und Drehung um die eigene Achse 6
      2.2.2. Die Ringe des Saturn 8
   2.3. Die Bahnen der Planeten 9
      2.3.1. Einleitung 9
      2.3.2. Umsetzung nach VRML 9
      2.3.3. Die Mondbahn 14
      2.3.4. Die Jupitermonde 15
   2.4. Der Rückwärtsgang 15
   2.5. Die Kameras 16
3. Das VRML-Ereignismodell 17
   3.1. eventIn und eventOut 17
   3.2. eventOut 18
   3.3. exposedField 18
4. Das External Authoring Interface (EAI) 18
   4.1. Konfiguration 19
   4.2. Programmieren mit dem EAI 19
   4.3. Referenz zum VRML-Browser 19
   4.4. Zugriff auf VRML-Knoten 19
5. Das Java-Programm 21
   5.1. Klassendiagramm 21
   5.2. Dokumentation 22
6. Literaturverzeichnis 27

1. Die Benutzerschnittstelle

Die Benutzerschnittstelle zum Sonnensystem-Modell ist ein im Webbrowser eingebettetes Java-Applet. Bei der Gestaltung der Benutzerschnittstelle waren folgende Punkte wichtig:

- Es sollten Funktionen zur Verfügung gestellt werden, die über die im VRML-Browser integrierten Funktionen hinausgehen, d.h. die Benutzerschnittstelle sollte einen Mehrwert bieten.
- Das Applet sollte nicht zu viel Raum auf dem Bildschirm einnehmen, also nicht zu sehr im Vordergrund stehen.

Da der zur Verfügung stehende Platz also recht beschränkt war, wurden lediglich folgende Funktionen implementiert:

- Anzeige des aktuellen Datums des Sonnensystem-Modells
- textuelle Eingabe eines Datums
- schnelles Vor- und Zurückspulen der Modellzeit
- direkte Auswahl eines Planetens
- Ein- und Ausschalten des Gitternetzes, der Orbit-Visualisierung und der Planetenachsen

Folgende Abbildung zeigt die Benutzerschnittstelle. Neben Funktionen zur Navigation in Zeit (Zeitanzeige und Setzen der Zeit, Schnelles Vor- und Zurückspulen) und Raum (Selektion eines Planeten) ist es auch möglich, bestimmte Hilfslinien (z.B. Anzeige der Planetenbahnen) ein- und auszuschalten. Planeten werden durch das Anklicken einer Grafik selektiert.

Abbildung 1: Die Benutzerschnittstelle

Nach der Initialisierung von Applet und VRML-Browser sieht man das gesamte Sonnensystem mit eingeschalteten Planetenbahnen und –achsen sowie einem Gitternetz zur Übersicht.

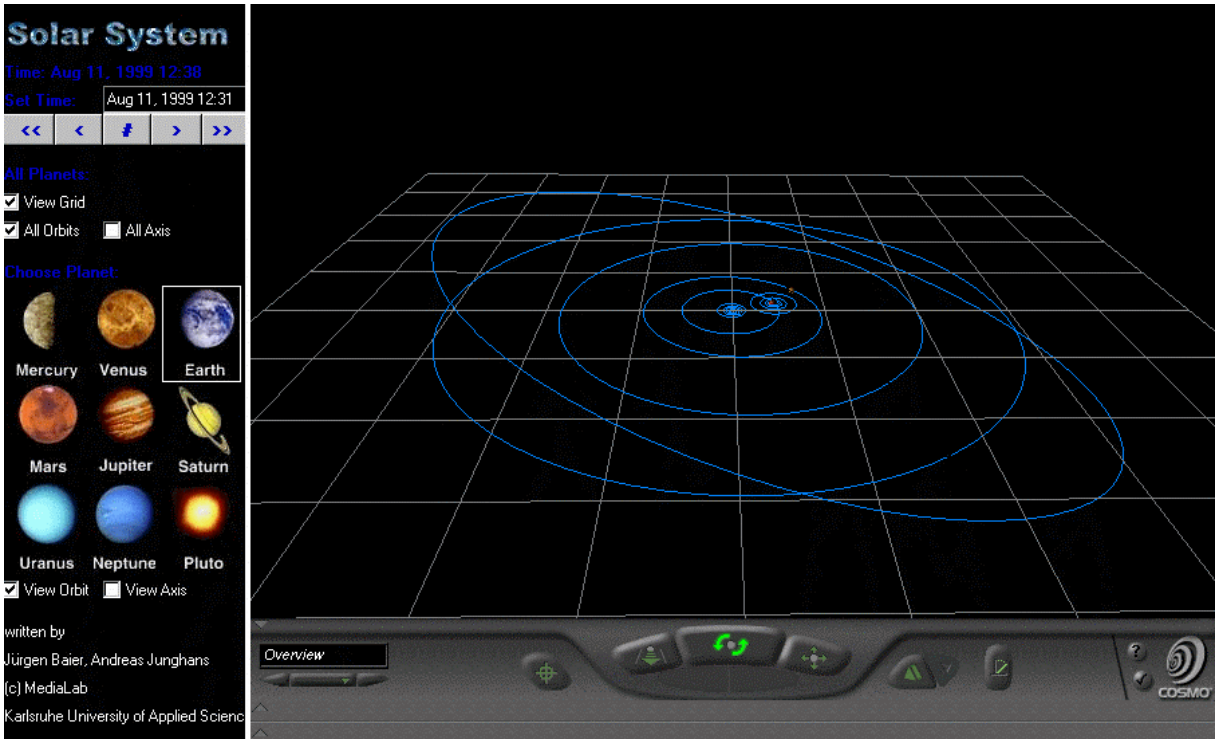


Abbildung 2: Übersicht Sonnensytem

Bei der direkten Anwahl eines Planeten zoomt der VRML-Browser direkt in eine nähere Ansicht aus einer interessanten Perspektive. Hier sieht man den Jupiter (mit einer detaillierten Textur):

Abbildung 3: Der Jupiter

Die Sonnensystem-Simulation arbeitet relativ genau. So ist es möglich, Planetenkonstellationen zu einem bestimmten Datum genau zu untersuchen. Die folgende Abbildung zeigt die Sonnenfinsternis am 11. August 1999 um 12:31. Man sieht, daß der Mond genau zwischen Sonne und Erde steht:

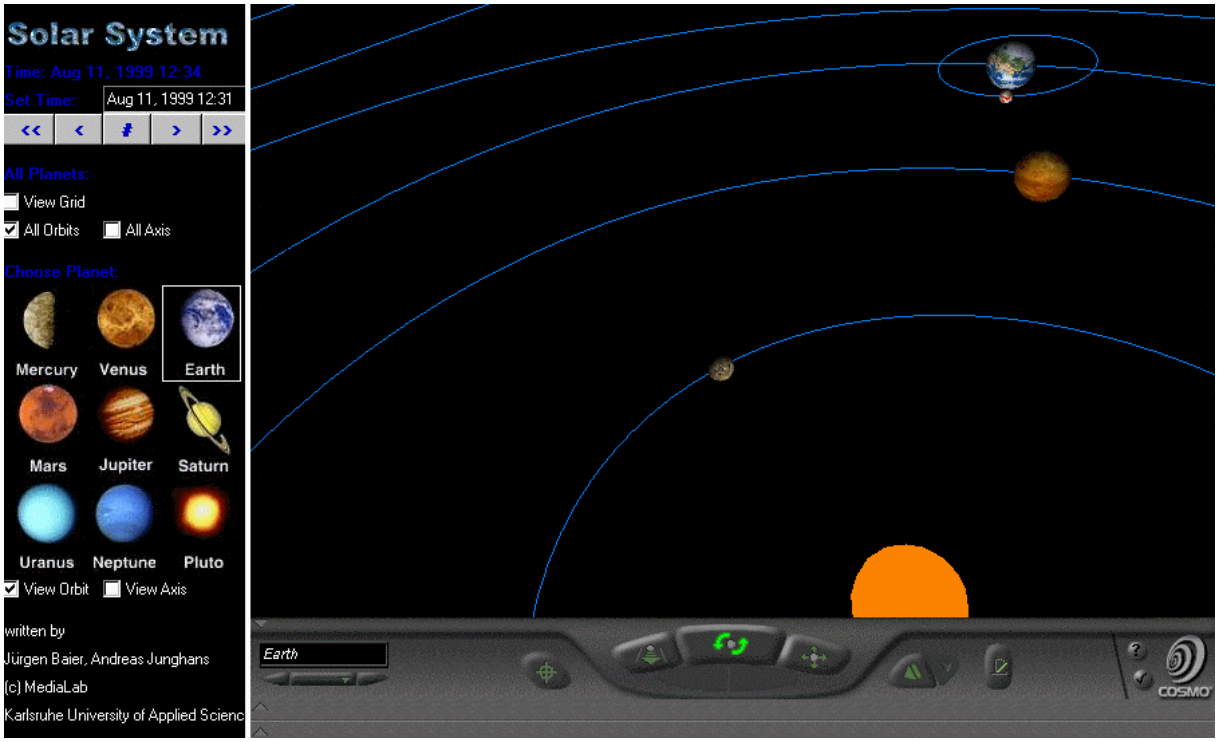


Abbildung 4: Die Sonnenfinsternis

Detaillierte Texturen aus dem NASA-Archiv tragen zu zusätzlichem Realismus bei:

Abbildung 5: Die Erde

2. Erstellung des VRML-Modells

Unser Ziel bei der Modellierung des Sonnensystems war, eine möglichst korrekte Darstellung der Planeten und Monde und ihrer Bewegungen zu erreichen. Es sollte zum Beispiel möglich sein, die Sonnenfinsternis vom 11.08.1999 anhand des Modells nachzuvollziehen.

2.1. Das Koordinatensystem

Das Modell verwendet das J2000-Koordinatensystem, wobei die Ekliptik in der XY-Ebene liegt. J2000 ist ein astronomischer Standard sowohl für räumliche Koordinaten als auch die Zeitmessung (Anzahl der Tage seit 1.1.2000 12:00 UTC). Eine Längeneinheit des Modells entspricht einer Astronomischen Einheit (AE), konkret wurde dafür der Wert von 149.598.000 km benutzt. Zur besseren Orientierung existiert ein Gitternetz in der XY-Ebene mit einem Abstand der einzelnen Linien von 10 AE.

Um eine zeitmäßig richtige Darstellung des Sonnensystems zu erreichen, sind alle VRML-Interpolatoren so ausgelegt, dass ihr Startwert für den 1.1.1970 korrekt ist. Dieses Datum entspricht der Zeit 0 im VRML-Standard. Wenn man nun 0 als Startzeit und „Echtzeit“ für die Intervalle aller *TimeSensor*-Knoten verwendet, erhält man eine Ansicht des Sonnensystems, die die tatsächlichen Verhältnisse zum aktuellen Zeitpunkt wiederspiegelt. Da die Animation in Echtzeit eine recht langweilige Sache ist, können die Startwerte und Intervalle durch das zur Steuerung verwendete Java-Applet in verschiedenen Stufen angepaßt werden (siehe Kapitel 1).

2.2. Die Planeten

2.2.1. Form, Oberfläche und Drehung um die eigene Achse

Im folgenden wird die Modellierung eines Planeten anhand der Erde Schritt für Schritt nachvollzogen. Am Ende steht genau der VRML-Code, der auch im fertigen Modell zu sehen ist.

Der einfachste Ansatz ist, den Planeten als einfarbige Kugel darzustellen:

```
Shape {
    appearance Appearance {
        material Material {
            ambientIntensity 0
            diffuseColor 0 0 1
            shininess 0.9
        }
    }
    geometry Sphere {
        radius 0.04263425
    }
}
```

Der Anteil ambienter Beleuchtung ist 0, da im freien Weltall keine Atmosphäre existiert, die das Licht streuen könnte. Die Farbe für die diffuse Beleuchtung ist blau - schließlich handelt es sich um die Erde - und die Oberfläche ist durch die Meere ziemlich glänzend. Der Radius der Erdkugel ist 6378 km, was umgerechnet in AE und 1000-fach vergrößert den Wert 0.043 ergibt. Die Vergrößerung wurde von uns verwendet, da sonst bei den riesigen Entfernungen nur winzige Pünktchen für die Planeten erscheinen würden, was zwar realistisch, aber nicht besonders ansehnlich ist.

Unsere Erde weist in diesem Stadium aber noch einige Unzulänglichkeiten auf. Der größte Schwachpunkt in der Optik ist die einfarbige Erscheinung. Um einen realistischeren Eindruck zu bekommen, muß eine Textur verwendet werden:

```
appearance Appearance {
    material Material {
        ...
    }
```

```
    texture ImageTexture {
        url "earthTexture.jpg"
    }
}
```

Die Textur stammt von [12], wie auch die Texturen für die anderen Planeten und den Erdmond. Die Texturen der Jupitermonde stammen von [10]. Das Aussehen unserer Erde ist nun schon fast perfekt, aber die Planeten sind in Wirklichkeit keine exakten Kugeln, sondern Ellipsoide mit dem größten Radius am Äquator:

```
DEF EarthEllipsoid Transform {
    scale 1 0.9967074 1
    children [

    Shape {
        ...
    }

] }
```

Damit hat unsere Erde den korrekten Polradius von 6357 km. Bei der Erde fällt der Unterschied zur Kugel nicht besonders auf, bei den größeren Planeten, z.B. Saturn, dagegen schon eher. Die Skalierung erfolgt entlang der Y-Achse, damit sie, bezogen auf die Textur, auch wirklich den Polabstand verkleinert. Später erfolgen dann einige Rotationen, die die Lage relativ zur Ekliptik endgültig festlegen.

Die Rotationsachse der Erde ist durch die Textur leicht zu erkennen, sofern man schon einmal eine Weltkarte gesehen hat. Bei anderen Planeten, z.B. dem Mars, hat man hier aber Schwierigkeiten. Deshalb bekommen die Planeten in unserem Modell eine sichtbare Rotationsachse verpasst:

```
DEF EarthRotationAxis Transform {
    children [

    Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0.5 0.5 0.5
            }
        }
        geometry Cylinder {
            radius 0.0042635
            height 0.170537
        }
    }

] }
```

Jetzt wird die Erde von einem grauen Zylinder durchbohrt, der doppelt so hoch wie ihr Durchmesser ist und dessen Radius ein Zehntel des Planetenradius beträgt. Allerdings liegt diese Achse in der Bahnebene und nicht, wie es sein sollte, 23,45° gegen sie geneigt. Dieser Mißstand wird durch zwei Rotationen korrigiert:

```
DEF EarthAxis Transform {
    rotation 1 0 0 -0.4092797096   # (-)23,45°
    children [

    DEF Earth Transform {
        children [

        DEF EarthOrientationCorrection Transform {
            rotation 1 0 0 1.5707963    # 90°
```

```
            children [

            DEF EarthRotationAxis Tranform {
                ...
            }

            DEF EarthEllipsoid Transform {
                ...
            }

        ] }

    ] }

] }
```

Die innere Rotation sorgt dafür, das die Rotationsachse senkrecht zur Bahnebene steht. Dadurch kann man in der äußeren Rotation direkt den Neigungswinkel ablesen. Die X-Achse als äußere Drehachse ist willkürlich gewählt, d.h. die Achse zeigt höchstwahrscheinlich nicht Richtung Polarstern. Eine korrekte Neigung wäre jedoch nur aufwändig zu realisieren gewesen und spielt für die optische Erscheinung des Modells keine Rolle. Das negative Vorzeichen vor dem Drehwinkel führt zu einer weitgehend richtigen Darstellung der Jahreszeiten. Bei positivem Winkel sind Sommer und Winter vertauscht.

An diesem Punkt haben wir eine recht ansehnliche, aber noch stillstehende Erde. Hier kommt der sinnlos erscheinende Knoten *Earth* aus obigem Code ins Spiel: Durch Veränderung seiner Rotation-Komponente kann eine Drehung der Erde um ihre eigenen Achse erreicht werden. Prinzipiell hätte man dazu auch den Knoten *EarthOrientationCorrection* benutzen können, hätte dann aber die X-Achse als Drehachse gehabt, was bei einer Ekliptik in der XY-Ebene weniger anschaulich ist. So können wir einen *OrientationInterpolator* verwenden, der eine gleichmäßige Drehung um die Z-Achse beschreibt:

```
DEF EarthAroundItself OrientationInterpolator {
    key [0, 0.33, 0.66, 1]
    keyValue [0 0 1 0, 0 0 1 2.1, 0 0 1 4.2, 0 0 1 0]
}
```

Zusätzlich wird ein *TimeSensor* benötigt, der die Drehung animiert:

```
DEF EarthDay TimeSensor {
    cycleInterval 86164.2
    loop TRUE
    startTime 0
}
```

Wie in Kapitel 2.1 beschrieben, erfolgt die Drehung ohne Eingriff des Java-Applets in Echtzeit, daher beträgt das Intervall des Sensors 86164.2 Sekunden, was einem Erdtag entspricht. Zuletzt sorgen zwei Ereignispfade dafür, dass alles miteinander verbunden wird:

```
ROUTE EarthDay.fraction_changed TO EarthAroundItself.set_fraction
ROUTE EarthAroundItself.value_changed TO Earth.rotation
```

Damit haben wir einen animierten Planeten in seinem lokalen Koordinatensystem. Wie dieser auf einer Bahn um die Sonne bewegt werden kann, zeigt das Kapitel 2.3.

2.2.2. Die Ringe des Saturn

Eine Spezialbehandlung war für die Saturnringe nötig, da VRML kein Basisobjekt „Scheibe mit Loch“ zur Verfügung stellt. Wir haben daher das Java-Programm *RingGenerator* geschrieben,

das unter Angabe von innerem und äußerem Radius und der Anzahl Punkte, die auf den beiden Begrenzungskreisen erzeugt werden sollen, einen Ring als *IndexedFaceSet* erstellt.

Die Farben der Ringe sind anhand verschiedener Photos experimentell gewählt. Wir haben nur die deutlich erkennbaren Ringe A bis C in das Modell aufgenommen und nur die Cassinische Teilung (zwischen Ring A und B), nicht aber die Enckesche Teilung (innerhalb von Ring A) berücksichtigt. Auch die Jupiter- und Uranusringe sind nicht enthalten, lassen sich aber bei Bedarf leicht ergänzen.

2.3. Die Bahnen der Planeten

2.3.1. Einleitung

Die Planeten folgen recht komplizierten Pfaden um die Sonne, woran vor allem ihre gegenseitige Anziehung schuld ist. Für unser Modell haben wir uns auf perfekte elliptische Bahnen beschränkt. Die verwendeten Daten (von [11] und [13]) erlauben eine korrekte Ansicht am 1.1.2000 und eine weitgehend richtige Darstellung im Abstand von +/- 50 Jahren davon.

Die elliptische Bahn eines Planeten um die Sonne läßt sich durch folgende Daten beschreiben:

Große Halbachse (a)	Die große Halbachse der Bahnellipse (nicht zu verwechseln mit dem größten Abstand von der Sonne!).
Exzentrizität (e)	Gibt an, wie „elliptisch“ die Bahn ist. $e = \sqrt{1 - \frac{b^2}{a^2}}$ a: große Halbachse, b: kleine Halbachse
Mittlere Länge (L)	Winkel, der zum Zeitpunkt, für den die Daten gelten, die Position des Planeten auf seiner Bahn darstellt.
Länge des aufsteigenden Bahnpunkts (o)	Winkel, der angibt, an welcher Stelle die Planetenbahn die Ekliptik von unten nach oben „durchstößt“.
Länge des Perihels (p)	Winkel, der die Stelle der Bahn angibt, an der der Planet der Sonne am nächsten ist.
Bahnneigung (i)	Neigung der Planetenbahn zur der Ekliptik.
Tägliche Bewegung (n)	Winkeländerung pro Tag, die die Bewegung des Planeten auf seiner Bahn beschreibt.

Tabelle 2-1: Die Elemente der Planetenbahnen

Diese Daten genügen, um die Planetenbewegung zu spezifizieren. Sie können auch für die Bahnen von Satelliten, z.B. des Mondes, verwendet werden, wobei sie sich dann auf den umkreisten Planeten statt auf die Sonne beziehen. Alle Daten gelten nur für einen bestimmten Zeitpunkt. In unserem Fall haben wir J2000-Daten verwendet, die für den 1.1.2000 12:00 (UTC) gelten. Je weiter man sich vom Bezugszeitpunkt entfernt, desto ungenauer werden die Elemente der Bahndaten. Man kann diesen Effekt teilweise kompensieren, indem man eine lineare tägliche Veränderung der Daten um einen bestimmten Betrag annimmt. Da sich aber für +/- 50 Jahre keine ohne weiteres sichtbaren Änderungen für das Modell ergeben, haben wir darauf verzichtet (Ausnahme: die Mondbahn, siehe Kapitel 2.3.3).

2.3.2. Umsetzung nach VRML

Im folgenden wird Schritt für Schritt die Bewegung des Mars um die Sonne modelliert. Tabelle 2-2 listet die benötigten Daten auf.

Große Halbachse (a)	1,52366231 AE
Exzentrizität (e)	0,09341233
Mittlere Länge (L)	355,45332°
Länge des aufsteigenden Bahnpunkts (o)	49,57854°
Länge des Perihels (p)	336,04084°
Bahnneigung (i)	1,85061°
Tägliche Bewegung (n)	0,524033035°

Tabelle 2-2: Die Bahndaten des Mars

Das grundlegende Problem bei der Umsetzung in VRML stellt die elliptische Form der Planetenbahnen dar. Es gibt keinen Interpolator oder sonstige Anweisungen, die eine solche Animation zur Verfügung stellen. Die in gewisser Hinsicht einfachste Lösung wäre, einfach die gesamte Bahn in Form eines *PositionInterpolators* auszudrücken, d.h. eine gewisse Anzahl Bahnpunkte zu berechnen, zwischen denen dann interpoliert wird. Nachteilig dabei ist, dass man für eine weiche Animation eine „Koordinatenwüste“ erhält, und das pro Planet und Satellit! Die VRML-Datei wird dadurch leicht auf 1 bis 2 MByte Größe aufgebläht. Wenn man außerdem noch einen „Rückwärtsgang“ vorsehen will (siehe Kapitel 2.4), verdoppelt sich dieser Wert noch einmal. Dieser Ansatz war für uns nicht akzeptabel, und wir haben nach einem eleganteren Weg gesucht.

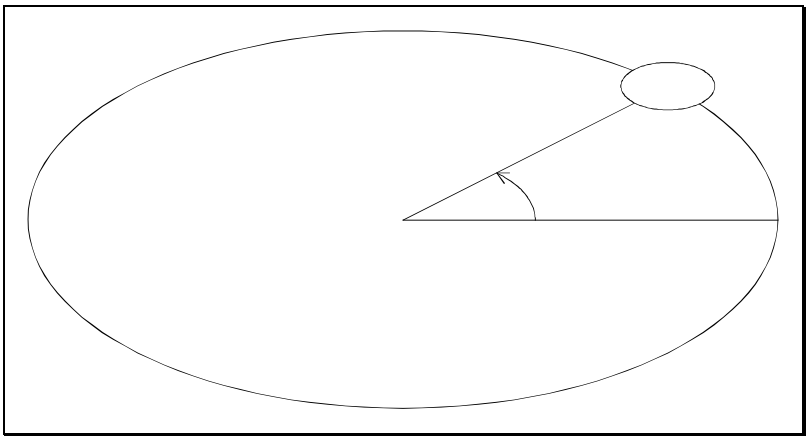


Abbildung 6: Deformation eines Planeten bei elliptischer Rotation

Unsere Grundidee besteht darin, eine gewöhnliche kreisförmige Rotation zu verwenden und durch eine darauf folgende Skalierung die gewünschte Ellipsenform zu erhalten. Allerdings wird dabei der Planet unschön verformt (Abbildung 6). Unsere Lösung dieses Problems besteht darin, den Planeten schon vor der Rotation zu deformieren, und zwar so, dass die nachfolgende Skalierung, mit der eine elliptische Bahn erreicht wird, diesen Effekt genau ausgleicht. Der VRML-Code dazu sieht, am Beispiel des Mars gezeigt, folgendermaßen aus:

```
DEF MarsAroundSun OrientationInterpolator {
    key [0, 0.33, 0.66, 1]
    keyValue [0 0 1 0, 0 0 1 2.1, 0 0 1 4.2, 0 0 1 0]
}

DEF MarsAxisCorrection OrientationInterpolator {
    key [0, 0.33, 0.66, 1]
    keyValue [0 0 1 0, 0 0 1 4.2, 0 0 1 2.1, 0 0 1 0]
}

DEF MarsOrbitScale Transform {
```

```
    scale 1.52366231 1.5170001 1
    children [

    DEF MarsOrbit Transform {
         # Rotation durch Interpolator MarsAroundSun
         children [

         # Reihenfolge der folgenden Transformationen laut
         # VRML 97-Standard:
         # zuerst Skalierung, dann Rotation, dann Translation
         DEF MarsFromSun Transform {
              translation 1 0 0
              # Rotation durch Interpolator MarsAxisCorrection
              scale 0.65631341 0.6591957 1
              children [

              DEF MarsAxis Transform {
                   ...
              }

         ] }

    ] }

] }

ROUTE MarsAroundSun.value_changed TO MarsOrbit.rotation
ROUTE MarsAxisCorrection.value_changed TO MarsFromSun.rotation
```

Die äußere Skalierung gibt die Werte für die große und die kleine Halbachse in AE wieder. Die kleine Halbachse läßt sich aus *a* und *e* berechnen (siehe Tabelle 2-1). Der *MarsOrbit*-Knoten dient zur Rotation um die Sonne, seine Rotation-Komponente wird durch den Interpolator *MarsAroundSun* gesetzt. Der eigentlich interessante Knoten ist *MarsFromSun*, da hier die „Intelligenz“ steckt. Zunächst wird die angesprochene Skalierung durchgeführt, die den Planeten „vordeformiert“. Erwartungsgemäß kommen hier genau die Kehrwerte der äußeren (Bahn-)Skalierung zum Einsatz. Würde man es dabei belassen, hätte man allerdings nicht viel erreicht, da die Rotation um die Sonne auch den verformten Planeten dreht. Dadurch passen die beiden Skalierungen dann nicht mehr zusammen, und man erhält ein recht seltsam anzuschauendes Ei. Deshalb wird nach der Skalierung der Planet genau entgegen seiner Bahnbewegung gedreht, bevor eine Translation ihn von der Sonne weg bewegt. Nach der Rotation um die Sonne ist der deformierte Himmelskörper genau so ausgerichtet, dass die äußere Skalierung ihn wieder in die richtige Form bringt und gleichzeitig seine Bahn zu einer Ellipse macht (siehe Abbildung 7).

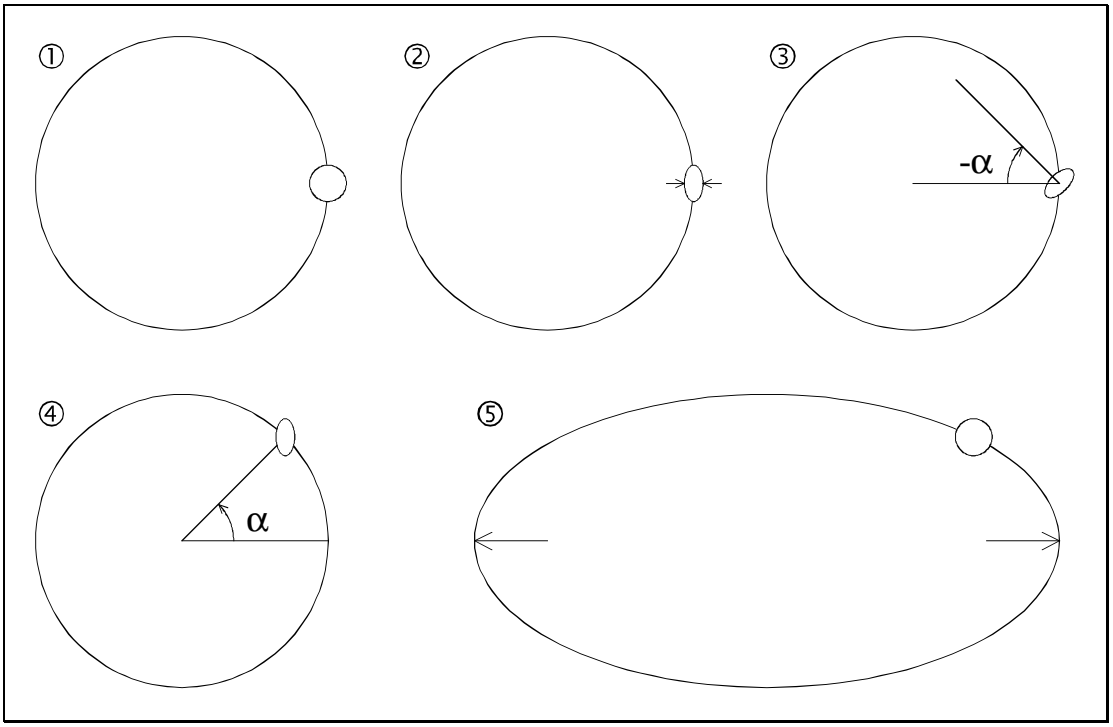


Abbildung 7: Ausgleich der Deformation

Durch die Drehung des Planeten entgegen seiner Bahnbewegung wird erfreulicherweise gleich ein weiteres Problem bei der Bewegung um die Sonne ausgeglichen: Die planetare Drehachse muß (annähernd) fix im Vergleich zum Sternenhintergrund bleiben. Würde sie die Rotation mitmachen, wäre zum Beispiel immer auf der gleichen Halbkugel Sommer! Durch die ausgleichende „Gegendrehung“, die aufgrund des Deformationsproblems erforderlich ist, wird gleichzeitig auch dieser Effekt kompensiert (daher auch der Name *MarsAxisCorrection* für den zweiten Interpolator).

Momentan befindet sich die Sonne noch im Mittelpunkt unserer Bahn, sie muß aber, von „oben“ betrachtet, im rechten Brennpunkt der Bahnellipse stehen:

```
DEF MarsOrbitFocus Transform {
    translation -0,1423288 0 0
    children [

    DEF MarsOrbitScale {
         ...
    }

] }
```

Der Betrag der Verschiebung errechnet sich als $a \cdot e$.

Nun fehlen nur noch die Drehung der Bahn um die Länge des Perihels und die Neigung gegenüber der Ekliptik:

```
DEF MarsOrbitInclination Transform {
    rotation 0.6484051 0.7612955 0 0.0322992    # 1,85061°
    children [

    DEF MarsOrbitRotation Transform {
         rotation 0 0 1 5.8650191   # 336,04084°
         children [

         DEF MarsOrbitFocus Transform {
              ...
```

```
        }
    ] }
] }
```

Die Drehachse für die Bahnneigung ergibt sich aus zwei Punkten, nämlich dem Brennpunkt, in dem die Sonne steht, und dem Punkt, an dem die Bahn die Ekliptik von unten nach oben „durchstößt“. Da die Sonne im Ursprung unseres Koordinatensystems liegt, kann direkt der Winkel *o* (siehe Tabelle 2-1) als Richtung der Drehachse verwendet werden, d.h. die X-Komponente der Achse ist gleich dem Cosinus, die Y-Komponente gleich dem Sinus von *o*. Da der Brennpunkt und der aufsteigende Knoten beide in der Ekliptik liegen, ist die Z-Komponente der Rotationsachse gleich 0.

Schließlich fügten wir noch eine grafische Darstellung der Bahnellipse ein, indem als weiterer Kindknoten von *MarsOrbitScale* ein Kreis gezeichnet wird (zu sehen im fertigem VRML-Code des Modells).

Um die Korrektheit der Planetenbahnen zu überprüfen, haben wir auf Basis von [11] ein Programm geschrieben, das die Bahn aus einzeln berechneten Punkten als *IndexedLineSet* in roter Farbe aufbaut, während wir einen weißen Kreis durch obige Transformationen geschickt haben. Das Ergebnis war eine weiß-rot gesprenkelte Kurve, die auch bei stark exzentrischen Ellipsen keine Trennung der beiden Varianten mehr zuließ, d.h. unsere Modellierung der Planetenbahnen kann als korrekt gelten. Zusätzlich haben wir die grafische Ausgabe des *Solar System Simulators ([7])* mit der optischen Erscheinung unseres Modells verglichen und konnten keine Unterschiede ausmachen.

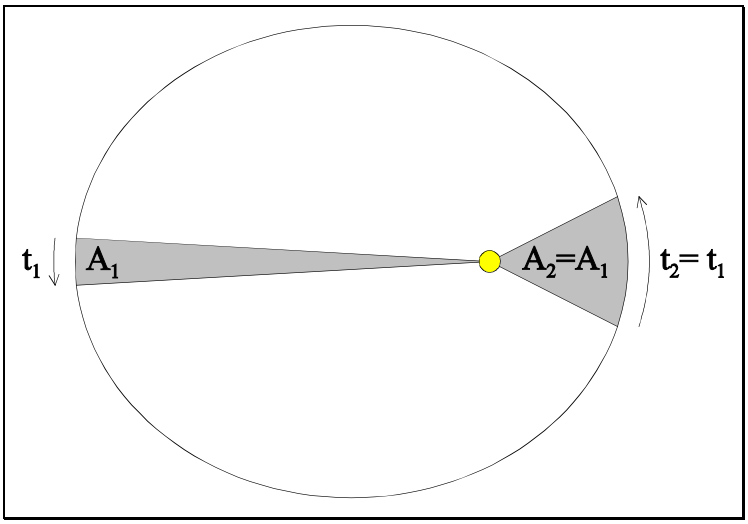


Abbildung 8: Das Zweite Keplersche Gesetz

An diesem Punkt ist die Planetenbahn bereits fertig modelliert, es fehlt allerdings noch die richtige Bewegung des Mars auf dieser Ellipse. Die im Beispielcode auf Seite 10 verwendete „naive“ Rotation ist aus mehreren Gründen kein Abbild der tatsächlichen Drehung. Erstens folgen die in bestimmten Zeiträumen zurückgelegten Winkel dem Zweiten Keplerschen Gesetz, wonach in gleichen Zeiträumen gleiche Flächen überstrichen werden, die sich aus dem Bogen auf der Bahn und der Verbindung seiner Endpunkte mit der Sonne ergeben (Abbildung 8). Zweitens beziehen sich die aus diesem Gesetz errechneten Winkel auf die Sonne als Drehpunkt, wobei der Radius entsprechend der elliptischen Bewegung variiert. Unsere Rotation erfolgt aber um den Mittelpunkt er Ellipse, da wir ansonsten noch einen zusätzlichen Interpolator für den Radius einführen müßten. Schließlich wird der Drehwinkel auch noch durch die Skalierung zur Ellipse (Knoten *MarsOrbitScale*) verfälscht. Es muß also ein *OrientationInterpolator* erzeugt werden, der das Zweite Keplersche Gesetz berücksichtigt und dabei die errechneten Winkel auf

den Mittelpunkt der Ellipse umrechnet und den Einfluß der Skalierung neutralisiert (siehe Abbildung 9). Das leistet das Java-Programm *OrbitGenerator*. Zusätzlich wird hier auch noch der Startwinkel des Interpolators auf den 1.1.1970 0:00 bezogen, um das in Kapitel 2.1 angesprochene „Echtzeit-Verhalten“ zu erzielen. Der Korrektur-Interpolator für die Achsstellung (siehe 2.3.2) wird bei dieser Gelegenheit gleich mit erzeugt, ebenso die entsprechenden Interpolatoren für den „Rückwärtsgang“ (siehe Kapitel 2.4). Kommandozeilen-Parameter für das Programm sind die Bahnelemente *n, L, p* und *e*. Zusätzlich gibt es den Schalter *-J2000*, der den Bezugszeitpunkt festlegt. Dieser Parameter ist nötig, da uns die Daten der Mondbahn nur in einem leicht von J2000 verschiedenen Zeitrahmen vorliegen (siehe [13]).

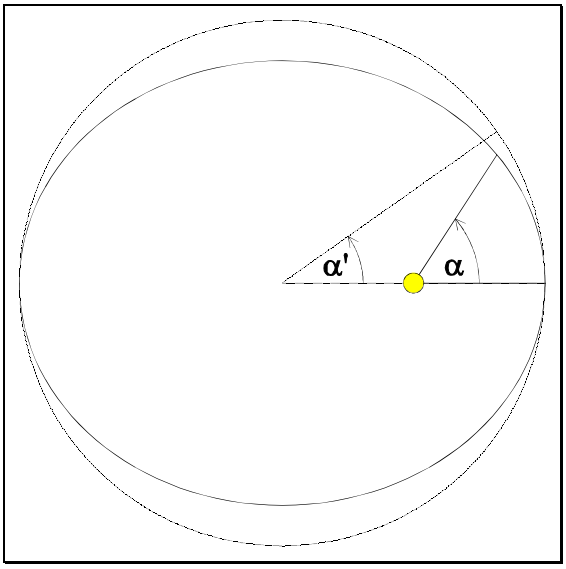


Abbildung 9: Gegebener (α) und benötigter Winkel (α')

2.3.3. Die Mondbahn

Die Bewegung des Mondes läßt sich prinzipiell genau wie die eines Planeten behandeln, nur dass hier die Erde den Bezugspunkt darstellt. Deshalb sind alle Knoten für die Mondbahn Kindknoten von *EarthFromSun*. Zusätzlich zu den bereits vorgestellten Transformationen muß die Rotation der Erdbahn um die Länge des Perihels ausgeglichen werden, da sich die Daten der Mondbahn auf den Fixsternhintergrund beziehen. Das erledigt der Knoten *MoonOrbitCorrection:*

```
DEF EarthFromSun Transform {
    translation 1 0 0
    scale 0.9999999 1.0001395 1
    children [

    DEF MoonOrbitCorrection Transform {
         rotation 0 0 1 -1.7967674  # -102,94719°
         children [

         DEF MoonOrbitInclination Transform {
              ...
         }

    ] }

] }
```

Eigentlich müßte auch die Neigung der Erdbahn gegen die Ekliptik ausgeglichen werden. Die gibt es tatsächlich, weil die „virtuelle“ Ekliptik des J2000-Koordinatensystems nicht vor dem

Fixsternhintergrund schwankt, wie es bei der realen der Fall ist. Dieser Neigungswinkel ist allerdings vernachlässigbar klein.

Eine Besonderheit, die speziell die Mondbahnbetrifft, ist ein Zyklus von 8,85 Jahren, in dem die Mondbahn sich einmal um die Erde dreht. Die „Nachstellung“ der Sonnenfinsternis vom 11.08.1999 ist nur möglich, wenn diese Rotation berücksichtigt wird, wohingegen andere Einflüsse, z.B. die Jupiter-Gravitation, hierfür vernachlässigt werden können. Wir verwenden folglich keine konstante Rotation der Bahn um die Länge des Perihels wie bei den Planeten (Knoten <*Planet*>*OrbitRotation*), sondern animieren sie. Um einen passenden *OrientationInterpolator* zu bekommen, haben wird das in Kapitel 2.3.2 vorgestellte Programm *OrbitGenerator* „mißbraucht“. Für *n* setzten wir den zum 8,85-Jahreszyklus passenden, pro Tag zurückgelegten Winkel ein, *L* bekam den Wert von *p* zugewiesen, um zum Bezugszeitpunkt (1.1.2000) die Länge des Perihels korrekt abzubilden. Die Parameter *p* und *e* des Programms sind für die Bahnrotation nicht von Belang bzw. stöhrend, deshalb werden sie auf 0 gesetzt. Dadurch findet weder das Zweite Keplersche Gesetz noch die Winkelkorrektur für die Ellipsenform statt. Die Überprüfung für das Datum 11.08.1999 ergibt für die Mittagszeit tatsächlich eine Stellung des Mondes zwischen Sonne und Erde.

Die große Halbachse der Bahn ist nicht maßstabsgerecht im Vergleich zu den Planetenbahnen, da durch die 1000-fache Vergrößerung der Planeten der Mond in diesem Fall irgendwo innerhalb der Erde kreisen würde. Die Entfernung von der Erde ist daher um den Faktor 50 vergrößert. Dieser Wert ist nach rein optischen Gesichtspunkten gewählt.

2.3.4. Die Jupitermonde

Wir haben leider keine kompletten Datensätze für die Bewegung der vier Galileischen Satelliten Jupiters im Internet finden können und auch nicht genug Zeit gehabt, um entsprechende astronomische Fachliteratur zu konsultieren, daher sind lediglich die Entfernung von Jupiter und die Bahnexzentrizität berücksichtigt. Wie beim Erdmond auch, mußten die Bahnen im Verhältnis zu denen der restlichen Planeten vergrößert werden, um die Jupiter-Satelliten überhaupt sichtbar zu machen. In diesem Fall wurde der Faktor 200 gewählt.

2.4. Der Rückwärtsgang

Als Steuerung des Modells schwebte uns eine Art Fernbedienung vor, mit der man, ähnlich einem Videorekorder, im Modell „spulen“ können sollte. Die verschiedenen Geschwindigkeiten in Richtung Zukunft stellten auch kein großes Problem dar, da hierfür nur die Felder der *TimeSensor*-Knoten richtig gesetzt werden müssen. Anders sieht es mit dem „Rückwärtsgang“ aus. Wir haben leider keine Möglichkeit gefunden, die *TimeSensor*-Knoten zum Rückwärtslaufen zu bewegen, ebensowenig die *OrientationInterpolator*-Knoten. Als einzige Lösung blieb, in das Programm *OrbitGenerator* auch die Generierung von Rückwärtsdrehungen einzubauen. Die Ereignisse dafür werden genau gleich behandelt wie diejenigen für die Vorwärtsdrehung. Der einzige Unterschied liegt darin, dass die Rückwärts-Interpolatoren jeweils ihre eigenen zugeordneten *TimeSensor*-Knoten haben, die normalerweise inaktiv sind. Bei Umschaltung der Richtung durch das Applet werden sie aktiviert und die Sensoren für die Vorwärtsdrehung deaktiviert. Die *ROUTE*-Anweisungen für einen Planeten sehen insgesamt folgendermaßen aus:

```
# vorwärts
ROUTE MarsDay.fraction_changed TO MarsAroundItself.set_fraction
ROUTE MarsAroundItself.value_changed TO Mars.rotation
ROUTE MarsYear.fraction_changed TO MarsAroundSun.set_fraction
ROUTE MarsAroundSun.value_changed TO MarsOrbit.rotation
ROUTE MarsYear.fraction_changed TO MarsAxisCorrection.set_fraction
ROUTE MarsAxisCorrection.value_changed TO MarsFromSun.rotation

# rückwärts
ROUTE MarsDayBackwards.fraction_changed TO
    MarsAroundItselfBackwards.set_fraction
ROUTE MarsAroundItselfBackwards.value_changed TO Mars.rotation
ROUTE MarsYearBackwards.fraction_changed TO
```

```
    MarsAroundSunBackwards.set_fraction
ROUTE MarsAroundSunBackwards.value_changed TO MarsOrbit.rotation
ROUTE MarsYearBackwards.fraction_changed TO
    MarsAxisCorrectionBackwards.set_fraction
ROUTE MarsAxisCorrectionBackwards.value_changed TO MarsFromSun.rotation
```

2.5. Die Kameras

Um eine vernünftige Navigation innerhalb des Modells zu ermöglichen, existiert eine Reihe von virtuellen Kameras in From von *Viewpoint*-Knoten. Für eine Übersicht gibt es die Kameras „Overview“ und „Inner Planets“, die das gesamte System bzw. die inneren Planeten bis einschließlich Mars von schräg oben zeigen. Daneben besitzt jeder Planet seine eigene Kamera, die ihm auf seiner Bahn folgt. Ziel war, eine Blickrichtung zu finden, die den Planeten im Zentrum hat, wobei der größte Teil der Oberfläche beleuchtet sein sollte und von schräg oben oder unten, je nach Bedarf, auf die Bahnebene geschaut wird.

Die Kamera eines Planeten durfte nicht in seinem lokalen Koordinatensystem liegen, denn dieses wird ja zum Ausgleich der Ellipsendeformation gedreht und skaliert. Um diesen Effekt nicht durch weitere Ereignispfade oder gar neue *OrientationInterpolator*-Knoten ausgleichen zu müssen, positionierten wir die Kameras im Koordinatensystem der Planetenbahn, aber schon innerhalb des *Transform*-Knotens, der für die Umkreisung der Sonne sorgt. Ein fertiger Blickpunkt sieht so aus:

```
# MarsOrbit wird für die Bewegung um die Sonne verwendet
DEF MarsOrbit Transform {
    children [

    DEF MarsViewScale Transform {
         scale 0.65631341 0.6591957 1
         children [

         DEF MarsViewTranslationX Transform {
              translation 1.52366231 0 0
              children [

              DEF MarsViewRotationZ Transform {
                   rotation 0 0 1 -0.785   # -45°
                   chilren [

                   DEF MarsViewRotationX Transform {
                        rotation 1 0 0 -0.5# -28,65°
                        children [

                        DEF MarsViewTranslationY Transform {
                             translation 0 -0.18 0
                             children [

                             DEF MarsView Viewpoint {
                                  position 0 0 0
                                  orientation 1 0 0 1.57
                                  description "Mars"
                             }

                        ] }

                   ] }

              ] }

         ] }

    ] }

] }
```

Zunächst wird die Kamera so ausgerichtet, dass der „Oben“-Vektor senkrecht zur Bahnebene steht und der Blick in Richtung der positiven Y-Achse geht. Anschließend sorgt *MarsViewTranslationY* dafür, dass ein gewisser Abstand zum Planeten gewonnen wird. Dieser Wert ist bei jedem Planeten entsprechend seiner Größe experimentell gewählt, so dass er möglichst gut im Bildschirmmittelpunkt zu sehen ist. *MarsViewRotationX* stellt die Kamera danach schräg, so dass leicht von oben geschaut wird. Bei dieser Einstellung ist der Planet allerdings nicht besonders schön beleuchtet, da die Sonne genau von links strahlt und die Hälfte des Planeten gar nicht „erwischt“. Deshalb dreht *MarsViewRotationZ* die Kamera ein Stück um die Z-Achse, so dass sie mehr aus Richtung der Sonne auf den Planeten zielt. Nun haben wir zwar gedanklich alle Drehungen und Verschiebungen auf das lokale Koordinatensystem des Planeten bezogen, aber wir befinden uns aus weiter oben genanntem Grund nicht darin! Deshalb verschiebt *MarsViewTranslationX* die Kamera zum Mars hin, nämlich um seine große Halbachse nach rechts (positive X-Achse). Schließlich ergibt sich noch das Problem, dass bei der Skalierung der Planetenbahn zur Ellipse in *MarsOrbitScale* auch die Kameraposition betroffen ist, wodurch der Blick am Planeten vorbei geht. Um dem entgegenzuwirken, wird durch *MarsViewScale* eine entgegengesetzte Skalierung durchgeführt. Da zwischen beiden Größenänderungen die Drehung um die Sonne liegt, können sie sich nicht vollkommen aufheben. Das fällt aber optisch nur bei stark exzentrischen Ellipsen auf, wie sie z.B. für Kometen typisch sind. Prinzipiell würden für X- und Y-Skalierung also ungefähre Werte reichen. In unserer Realisierung haben wir genau die verwendet, die auch bei der Vordeformation der Planeten zum Einsatz kommen, nämlich die Kehrwerte der äußeren Skalierung.

Im Zusammenhang mit den *Viewpoint*-Knoten existiert leider ein lästiger Bug im CosmoPlayer 2.1. Bei den Planeten-Kameras wandert der Planet langsam aber sicher aus der Bildmitte aus. Das liegt nicht an unseren Transformationen, da ein erneutes Aktivieren des gleichen Ansichtspunkts den Planeten wieder in die Mitte rückt. Offensichtlich akkumulieren sich in diesem Browser bei animierten Ansichten Rundungsfehler o.ä., so dass die Kameraposition mit der Zeit immer mehr „driftet“. Zwischenzeitlich schwebte uns vor, diesen Fehler dadurch zu korrigieren, dass das Applet periodisch die Kameraposition auf genau den Ansichtspunkt setzt, der bereits aktiv ist. Leider wären dadurch aber Drehungen und Bewegungen relativ zur aktuellen Ansicht unmöglich geworden, so dass wir die Idee verwerfen mussten.

3. Das VRML-Ereignismodell

In einer VRML-Welt sind verschiedene Objekte in Form einer hierarchischen Knotenstruktur angeordnet. Bei rein statischen Welten müssen die Objekte nicht miteinander kommunizieren. Oft will man jedoch Bewegungen anzeigen oder eine Interaktion mit dem Benutzer ermöglichen. Dazu verwendet man das VRML-Ereignismodell.

3.1. eventIn und eventOut

Sieht man sich das Ereignismodell näher an, stößt man auf zwei Methoden (im objektorientierten Sinn):

- eventIn und
- eventOut.

Mit eventOut werden Ereignisse ausgegeben und mit eventIn empfangen. Mit diesen beiden Methoden ist es möglich, den inneren Zustand des Knotens zu ändern. Ansonsten hat man keine Möglichkeit, diesen Zustand – etwa durch direkten Zugriff auf Felder – zu ändern. In einem Transform-Knoten sind eventIn und eventOut so definiert:

```
Transform {
    eventIn  MFNode       addChildren
    eventIn  MFNode       removeChildren
    ...
}
```

Die Methoden, auf die man von außen zugreifen kann, sind also addChildren und removeChildren. Der Datentyp MFNode zeigt an, daß hier ein Multifield-VRML-Knoten als Parameter erwartet wird. Doch Vorsicht: VRML kennt keine Methodenaufrufe mit

Parameterübergaben, wie bei objektorientierten Programmiersprachen. In VRML können Objekte nur über Ereignisse miteinander kommunizieren. Ein Ereignis besteht aus

- einem Wert,
- einem Zeitstempel und
- einem Aufruf.

Ein Wert muß stets einen bestimmten Datentyp haben. In obigem Beispiel ist der Datentyp MFNode. Jedes VRML-Ereignis trägt einen Zeitstempel (darauf hat man keinen Einfluß, es wird einfach der entsprechende Zeitwert übergeben). Die eventIn-Methode des Empfänger-Objekts wird nun jedesmal aufgerufen, wenn sich ein bestimmter Wert beim Sender-Objekt ändert.

3.2. eventOut

Ein eventOut wird von einem Objekt unter bestimmten Umständen generiert, z.B. wenn sich ein bestimmter Wert geändert hat oder einfach wenn die VRML-Zeit ein Zeitintervall fortgeschritten ist:

```
eventOutSFTime        time
```

Ein Ereignis kann jetzt auf eine eventIn-Methode gerouted werden:

```
ROUTE MercuryAroundItself.value_changed TO Mercury.rotation
```

Hier wird also ein eventOut vom MercuryAroundItself-Objekt gesendet, wenn sich der entsprechende Wert geändert hat (value_changed). Der Empfänger dieses Ereignisses ist die rotation-Methode (eventIn) des Mercury-Objekts.

3.3. exposedField

Es gibt auch spezielle Felder, die eventIn, eventOut und field vereinen. Diese Felder werden exposedFields genannt. Sie bestehen aus

- eventIn
- eventOut
- field

In einem Transform-Knoten gibt es ein exposedField "children":

```
Transform {
    exposedField  MFNode       children []
    ...
}
```

Dieses exposedField ist nun einfach eine verkürzte Schreibweise für:

```
Transform {
    field             MFNode       children []
    eventIn       MFNode       set_children
    eventOut      MFNode       children_changed
}
```

4. Das External Authoring Interface (EAI)

Das External Authoring Interface (EAI) ermöglicht die Kontrolle einer VRML-Welt über ein Java-Applet. Im Sonnensystem der Studienarbeit ist so unter anderem eine direkte Anwahl der Planeten und die Steuerung der Rotationsgeschwindigkeit möglich. Außerdem wird ständig das Datum der gerade aktuellen Planetenkonstellation angezeigt. Das Datum der VRML-Welt kann über eine Texteingabe sowie über eine Vor-/Zurückspulfunktion eingestellt werden.

Diese Java-VRML-Kommunikation wird über die Plugin-Schnittstelle des Web-Browsers ermöglicht. Leider gibt es hier – abhängig von der Web-Browser/VRML-Browser-Kombination – verschiedene Probleme.

Die VRML-EAI-FAQ (http://www.frontiernet.net/~imaging/eaifaq.html) beschreibt diverse mögliche Konfigurationen, zum Teil mit eher exotischer Hard- und Software.

4.1. Konfiguration

Nach verschiedenen Versuchen hat sich die Kombination Internet Explorer 5 und Cosmoplayer 2.1 (unter Windows NT4.0SP3) am stabilsten erwiesen. Bei der Cosmoplayer-Installation werden die notwendigen Umgebungsvariablen automatisch gesetzt. Für die Entwicklung von EAI-Applets muß mit den EAI-Klassen vom Cosmoplayer compiliert werden. Diese Klassen findet man im Cosmoplayer-Programmverzeichnis unter dem Namen

- npcosmop21.zip oder
- npcosmop21.jar.

Die Java-Version ist dabei relativ egal, gute Ergebnisse wurden mit Java 1.2 erzielt. Die Swing-Klassen können leider nicht verwendet werden, weil zur Zeit kein Web-Browser diese Klassen implementiert hat und das EAI nicht mit dem entsprechenden Browser-Plugin von Sun funktioniert.

4.2. Programmieren mit dem EAI

Wurde das VRML-Plugin korrekt installiert, gibt es keine nennenswerten Probleme beim Entwickeln mehr. Leider sind im Netz fast keine funktionsfähigen EAI-Beispiele zu finden. Die Website von Yong Bing (http://www.comp.nus.edu.sg/~khooyb/) bietet allerdings einige gut kommentierte und funktionsfähige Applets, die das EAI verwenden.

4.3. Referenz zum VRML-Browser

Die Referenz zum VRML-Browser wird durch

```
Browser browser = Browser.getBrowser(this);
```

hergestellt. Leider tritt dabei (vor allem beim Netscape Navigator) oft das Problem auf, daß das Java-Applet vor dem VRML-Modell initialisiert wird. Die Folge ist, daß das Applet (auch bei einem Reload) keine Referenz zum VRML-Browser mehr herstellen kann. Auf der Cosmo-Website (http://www.cai.com/cosmo/) wird folgende Lösung empfohlen:

```
Browser browser = null;
for (int count = 0; count < 10; count++) {
    try {
        try { Thread.sleep(200); } catch (InterruptedException ignored) { }
        browser = Browser.getBrowser(this);
    }
    catch (Exception e) { }  // getBrowser() can throw NullBrowserException
    if (browser != null) break;
    Browser.print("browser was null, trying again...");
}
if (browser == null) {
    throw new Error("Failed to get the browser after 10 tries!");
}
Browser.print("Got browser!");
```

Allerdings funktioniert auch diese Lösung nicht immer, so daß man gelegentlich gezwungen wird, den Web-Browser neu zu starten.

4.4. Zugriff auf VRML-Knoten

Ein VRML-Knoten hat den Datentyp Node. Also muß zunächst deklariert werden:

```
Node gridNode = null;
```

Die Initialisierung des Knotes erfolgt mit

```
gridNode = browser.getNode("Grid");
```

Wie man sieht, wird hier einfach ein String des DEFinierten Knotens übergeben. Dieser Knoten sieht dann im VRML-File so aus:

```
DEF Grid Transform {
    scale 10 10 10
    ...
}
```

Natürlich will man mit dem Grid-Knoten auch etwas anfangen können. Dazu werden die Events des Knotens deklariert:

```
EventInSFVec3f gridScaleIn = null;
EventOutSFVec3f gridScaleOut = null;
```

Um diese Events benutzen zu können, müssen sie über den Grid-Knoten initialisiert werden:

```
gridScaleIn = (EventInSFVec3f) gridNode.getEventIn("set_scale");
gridScaleOut = (EventOutSFVec3f) gridNode.getEventOut("scale_changed");
```

Das Grid wird nicht über addChildren und removeChildren ein- und ausgeschalten sondern über eine Skalierung, so daß das Gitter in der Sonne verschwindet. Dazu muß der neue Wert des scale-Feldes explizit gesetzt werden. Folgender Code schaltet das Grid ein:

```
float gridScaleValue[] = new float[3];
gridScaleValue[0] = Float.valueOf("10.0").floatValue();
gridScaleValue[1] = Float.valueOf("10.0").floatValue();
gridScaleValue[2] = Float.valueOf("10.0").floatValue();
gridScaleIn.setValue(gridScaleValue);
```

Soll das Grid ausgeschalten werden, müssen die float-Werte auf z.B. 0.01 gesetzt werden.

Das Ein- und Ausschalten der Orbits wurde mit addChildren und removeChildren realisiert. Dazu müssen lediglich die entsprechenden Events deklariert werden:

```
EventInMFNode mercuryOrbitCircleAddChildren = null;
EventInMFNode mercuryOrbitCircleRemoveChildren = null;
```

Diese Events werden über

```
mercuryOrbitCircleAddChildren
    = (EventInMFNode) mercuryOrbitCircleNode.getEventIn("addChildren");
mercuryOrbitCircleRemoveChildren
    = (EventInMFNode) mercuryOrbitCircleNode.getEventIn("removeChildren");
```

initialisiert. Eingeschalten wird der Orbit mit

```
mercuryOrbitCircleAddChildren.set1Value(0, circleNode);
```

Um den Orbit auszuschalten muß mit removeChildren gearbeitet werden

```
mercuryOrbitCircleRemoveChildren.set1Value(0, circleNode);
```

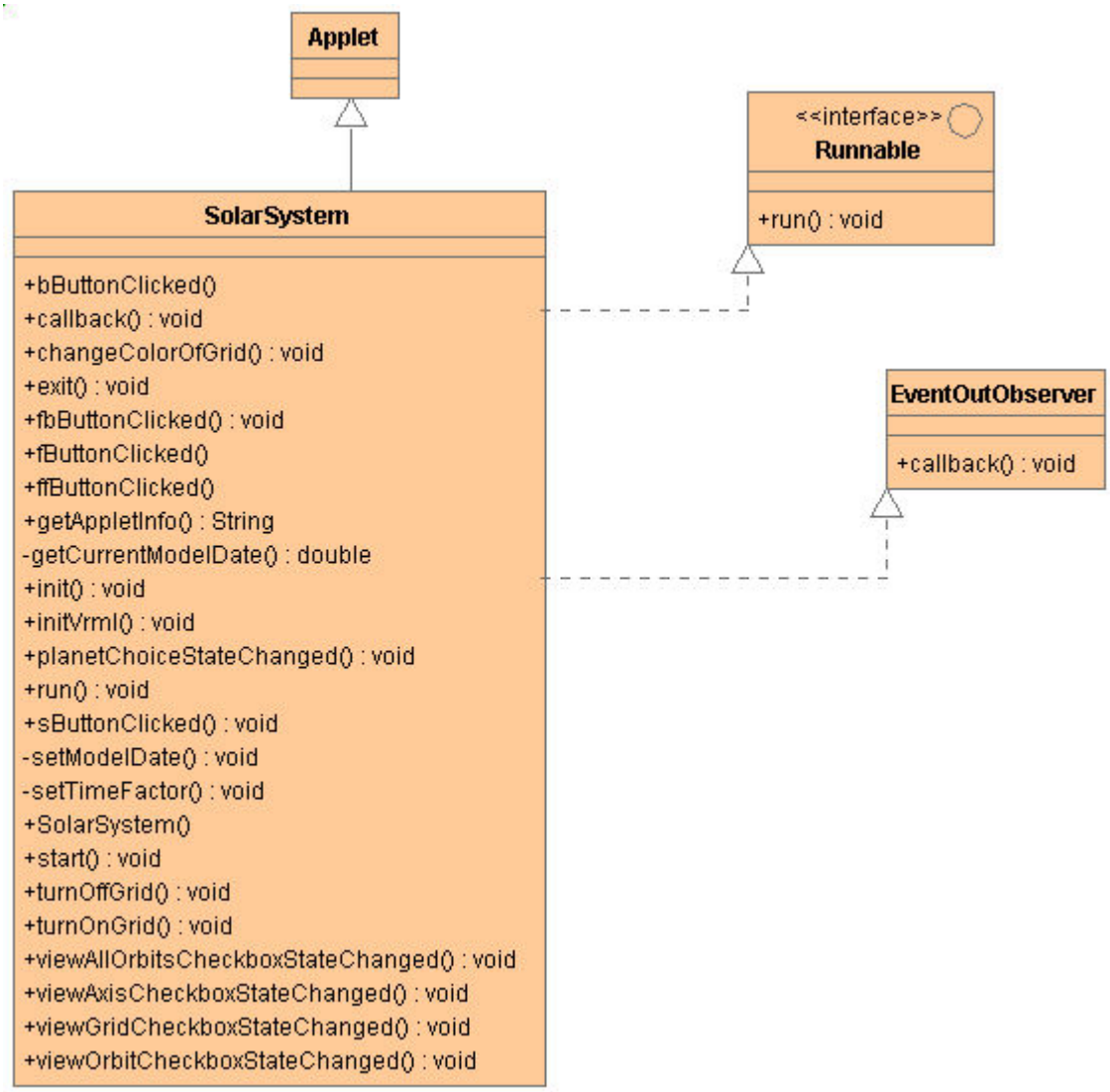
Wichtig beim Zugriff auf VRML-Knoten und deren Events ist nur, die korrekten Datentypen (Node, EventInMFNode, ...) zu verwenden und sich eine Standard-Schreibweise anzugewöhnen, um nicht die verschiedenen Eventarten zu verwechseln.

5. Das Java-Programm

Das Applet besteht aus einer public-Klasse (SolarSystem) und mehreren Inner Classes. Details zu den Methoden sind im Abschnitt Dokumentation zu finden. Anzumerken ist noch, daß einige Methoden "leer" implementiert wurden bzw. obsolet sind. So ist z.B. die Implementierung von EventOutObserver (um Callbacks von der VRML-Welt zu bekommen) leer, da es Probleme mit dem Internet Explorer gab (Absturz mit allgemeiner Schutzverletzung).

5.1. Klassendiagramm

(Erstellt mit MagicDraw UML)



5.2. Dokumentation

(Generiert mit MagicDraw UML)

Solar System

Generated at: Sat, Feb 26, 2000
15:25:30

TABLE OF CONTENTS

REPORT BODY

Use case report

Class report
Data package
- SolarSystem class
- Runnable class
- EventOutObserver class
- Applet class

Component View package
Model dictionary

SUMMARY

REPORT BODY

Use case report

Class report

package Data

class SolarSystem

Superclasses:
- Data::Applet

Interfaces:
- Data::Runnable
- Data::EventOutObserver

Inner classes:

class PlanetsCanvas

Stereotype: <<GUI>>
Documentation: *Class for viewing the planets image map.*
Operations:
public void **setPlanet()**
public void **paint()**
public void **update()**

class TitleCanvas

Stereotype: <<GUI>>
Documentation: *Class for viewing the title graphics.*

Attributes:
private **titleImage** : Image = null
private **sizeSet** : boolean = false
Operations:
public **TitleCanvas ( )**
public void **paint ( )**

class SolsysBodyTime

Documentation: *Class for handling of a time sensor*
Operations:
public **SolsysBodyTime ( )**
Documentation: *Constructs a SolsysBodyTime object.*
public double **getTime ( )**
Documentation: *Gets the current time.*
public void **stop ( )**
Documentation: *Stops the time sensor.*
public void **setTime ( )**
Documentation: *Sets the startTime and cycleInterval fields according to the given date settings and enables the sensor. The sensor must be disabled prior to calling this method. The current date is given as parameter to avoid differences between the positions of the bodies due to latency while adjusting their TimeSensors. The date parameters are given in seconds since the epoch.*

class SolsysBody

Documentation: *Class for handling of a cellestial body*
Operations:
public **SolsysBody ( )**
Documentation: *Constructs a SolsysBody object.*
public String **getName ( )**
Documentation: *Returns the name of this body.*
public boolean **isAxisVisible ( )**
Documentation: *Determines if the body's axis is visible. Children are not involved.*
public void **setAxisVisible ( )**
Documentation: *Shows/hides the body's axis. Children are not involved.*
public void **setAllAxesVisible ( )**
Documentation: *Shows/hides all the axes of the body and all of it's (sub)children.*
public boolean **isOrbitVisible ( )**
Documentation: *Determines if the body's orbit is visible. Children are not involved.*
public void **setOrbitVisible ( )**
Documentation: *Shows/hides the body's orbit. Children are not involved.*
public void **setAllOrbitsVisible ( )**
Documentation: *Shows/hides all the orbits of the body and all of it's (sub)children.*
public void **viewThis ( )**
Documentation: *Changes the Viewpoint to this body.*
public void **addChild ( )**
Documentation: *Adds a child.*
public SolsysBody **findChild ( )**
Documentation: *Finds a child by a given name. Subchildren are not searched.*
public double **getTime ( )**
Documentation: *Gets the current time.*
public void **stop ( )**
Documentation: *Stops the time sensor.*
public void **stopAll ( )**

Documentation: *Stops the time sensors of the body and all of it's (sub)children.*
public void **setTime()**
Documentation: *Sets the startTime and cycleInterval fields according the given date settings and enables the sensor. The sensor must be disabled prior to calling this method. The current date is given as parameter to avoid differences between the positions of the bodies due to latency while adjusting their TimeSensors. The date parameters are given in seconds since the epoch.*
public void **setTimeAll()**
Documentation: *Sets the time of the time sensors of the body and all of it's (sub)children.*

Operations:
public **SolarSystem()**
Documentation: *constructor*
public void **start()**
public void **initVrml()**
public String **getAppletInfo()**
public void **run()**
public void **viewAllOrbitsCheckboxStateChanged()**
public void **exit()**
public void **callback()**
public void **changeColorOfGrid()**
public void **viewGridCheckboxStateChanged()**
public void **turnOnGrid()**
public void **turnOffGrid()**
public void **viewAxisCheckboxStateChanged()**
public void **viewOrbitCheckboxStateChanged()**
public void **fbButtonClicked()**
public **bButtonClicked()**
public void **sButtonClicked()**
public **fButtonClicked()**
public **ffButtonClicked()**
private void **setModelDate()**
private void **setTimeFactor()**
private double **getCurrentModelDate()**
public void **planetChoiceStateChanged()**
public void **init()**
Documentation: *Initialization of the user interface.*

class Runnable
Stereotype: <<interface>>
Operations:
public void **run()**

class EventOutObserver
Operations:
public void **callback()**

class Applet

package Data::Component View

A

addChild operation from class SolarSystem.SolsysBody Adds a child.
Applet class
B

bButtonClicked operation from class SolarSystem
C

callback operation from class SolarSystem
callback operation from class EventOutObserver
changeColorOfGrid operation from class SolarSystem
Component View package
D

Data package
E

EventOutObserver class
exit operation from class SolarSystem
F

fbButtonClicked operation from class SolarSystem
fButtonClicked operation from class SolarSystem
ffButtonClicked operation from class SolarSystem
findChild operation from class SolarSystem.SolsysBody Finds a child by a given name. Subchildren are not searched.
G

getAppletInfo operation from class SolarSystem
getCurrentModelDate operation from class SolarSystem
getName operation from class SolarSystem.SolsysBody Returns the name of this body.
getTime operation from class SolarSystem.SolsysBodyTime Gets the current time.
getTime operation from class SolarSystem.SolsysBody Gets the current time.
I

init operation from class SolarSystem Initialization of the user interface.
initVrml operation from class SolarSystem
isAxisVisible operation from class SolarSystem.SolsysBody Determines if the body's axis is visible. Children are not involved.
isOrbitVisible operation from class SolarSystem.SolsysBody Determines if the body's orbit is visible. Children are not involved.
P

paint operation from class SolarSystem.TitleCanvas
paint operation from class SolarSystem.PlanetsCanvas
planetChoiceStateChanged operation from class SolarSystem
PlanetsCanvas class from class SolarSystem Class for viewing the planets image map.
R

run operation from class SolarSystem
run operation from class Runnable
Runnable class

S

sButtonClicked operation from class SolarSystem
setAllAxesVisible operation from class SolarSystem.SolsysBody Shows/hides all the axes of the body and all of it's (sub)children.
setAllOrbitsVisible operation from class SolarSystem.SolsysBody Shows/hides all the orbits of the body and all of it's (sub)children.
setAxisVisible operation from class SolarSystem.SolsysBody Shows/hides the body's axis. Children are not involved.
setModelDate operation from class SolarSystem
setOrbitVisible operation from class SolarSystem.SolsysBody Shows/hides the body's orbit. Children are not involved.
setPlanet operation from class SolarSystem.PlanetsCanvas
setTime operation from class SolarSystem.SolsysBodyTime Sets the startTime and cycleInterval fields according to the given date settings and enables the sensor. The sensor must be disabled prior to calling this method. The current date is given as parameter to avoid differences between the positions of the bodies due to latency while adjusting their TimeSensors. The date parameters are given in seconds since the epoch.
setTime operation from class SolarSystem.SolsysBody Sets the startTime and cycleInterval fields according the given date settings and enables the sensor. The sensor must be disabled prior to calling this method. The current date is given as parameter to avoid differences between the positions of the bodies due to latency while adjusting their TimeSensors. The date parameters are given in seconds since the epoch.
setTimeAll operation from class SolarSystem.SolsysBody Sets the time of the time sensors of the body and all of it's (sub)children.
setTimeFactor operation from class SolarSystem
sizeSet attribute from class SolarSystem.TitleCanvas
SolarSystem operation from class SolarSystem constructor
SolarSystem class
SolsysBody operation from class SolarSystem.SolsysBody Constructs a SolsysBody object.
SolsysBody class from class SolarSystem Class for handling of a cellestial body
SolsysBodyTime class from class SolarSystem Class for handling of a time sensor
SolsysBodyTime operation from class SolarSystem.SolsysBodyTime Constructs a SolsysBodyTime object.
start operation from class SolarSystem
stop operation from class SolarSystem.SolsysBodyTime Stops the time sensor.
stop operation from class SolarSystem.SolsysBody Stops the time sensor.
stopAll operation from class SolarSystem.SolsysBody Stops the time sensors of the body and all of it's (sub)children.

T

TitleCanvas operation from class SolarSystem.TitleCanvas
TitleCanvas class from class SolarSystem Class for viewing the title graphics.
titleImage attribute from class SolarSystem.TitleCanvas
turnOffGrid operation from class SolarSystem
turnOnGrid operation from class SolarSystem

U

update operation from class SolarSystem.PlanetsCanvas
V

viewAllOrbitsCheckboxStateChanged operation from class SolarSystem
viewAxisCheckboxStateChanged operation from class SolarSystem
viewGridCheckboxStateChanged operation from class SolarSystem
viewOrbitCheckboxStateChanged operation from class SolarSystem
viewThis operation from class SolarSystem.SolsysBody Changes the Viewpoint to this body.

SUMMARY

```
Total packages reported:     2
Total classes reported:      8
Total attributes reported:      2
Total operations reported:      51
```

6. Literaturverzeichnis

[1]	The Virtual Reality Modeling Language, Committee Draft International Standard ISO/IEC 14772: http://www.web3d.org/technicalinfo/specifications/vrml97/index.htm
[2]	VRML 2.0 EAI FAQ: http://www.frontiernet.net/~imaging/eaifaq.html
[3]	Cosmo Website: http://www.cai.com/cosmo/
[4]	Website von Yong Bing: http://www.comp.nus.edu.sg/~khooyb/academic/vrml/contents_of_vrml.html
[5]	Sonnensystem-Texturen: http://www.lancs.ac.uk/postgrad/thomasc1/render/maps.htm
[6]	comp.lang.vrml.faq: http://home.hiwaay.net/~crispen/vrmlworks/faq/index.html
[7]	Solar System Simulator: http://space.jpl.nasa.gov/
[8]	Interactive Universe: http://iuniverse.gsfc.nasa.gov/iuniverse/
[9]	Mars Pathfinder VRML: http://mpfwww.jpl.nasa.gov/vrml/vrml.html
[10]	Björn Jónsson's homepage: http://www.mmedia.is/~bjj/
[11]	Approximate astronomical positions: http://www.stargazing.net/kepler/
[12]	Texturen: http://reality.sgi.com/sambo/Oobe/CyberAstronomy/intro.html
[13]	How to compute planetary positions: http://hotel04.ausys.se/pausch/comp/ppcomp.html