Soya3D Tutorial

Markus Raab
ich@markus-raab.org

Patrick Sabin
patricksabin@gmx.at

28. März 2008

Die folgende Arbeit ist eine Einführung in Soya3D, eine objektorientierte 3D-Engine für die Programmiersprache Python. Soya3D ist nicht nur leistungsfähig, sondern auch einfach zu verwenden. So ist es möglich, dass mit verhältnismäßig geringem Zeitaufwand beeindruckende Ergebnisse erziehlt werden, was auch unter dem Begriff Rapid-Prototyping subsumiert werden kann. Es wird jedoch nicht nur auf Verwendung von Soya3D eingegangen, sondern auch Vorteile und Nachteile zu anderen Engines betrachtet, die Besonderheiten beleuchtet und Einsatzgebiete aufgezeigt. Außerdem werden Technologien und Programme beschrieben, die üblicherweise in Zusammenhang mit Soya3D stehen.

Inhaltsverzeichnis

1 Einführung

Soya3D ist eine Bibliothek für Python [21][24][25][26][28][27] welche vor allem versucht im Stil, Eleganz und Avantgarde ähnlich Python zu bestechen. Damit können 3D Applikationen vollständig und in guter Performance in Python geschrieben werden. Dieses Tutorial zeigt, dass es neben leichter Erlernbarkeit auch vollständig die Features implementiert, die von einer 3D Engine erwartet werden können.

Dieses Tutorial wendet sich an interessierte, auch unerfahrene, Python Programmierer die ihre ersten 3D Applikationen schreiben wollen, als auch an Leute die eine andere, einfache Weise der 3D Programmierung kennenlernen wollen.

Neben syntaktischen Erläuterungen erklärt dieses Kapitel kurz die Lizenz und führt den Leser in Soya3D ein. Das anschließende Kapitel liefert fundierte unverzichtbare Grundlagen für jedes Soya3D Programm auf welche alle anderen Kapitel aufbauen.

Die Mathematik von Soya3D ist zwar als optional anzusehen, ohne diese Werkzeuge werden sie allerdings schnell an die Grenzen jeder Grafikengine, und auch Soya3D stoßen, Grundschulmathematik wird hier allerdings vorausgesetzt.

Das Kapitel Verwaltung von Daten erklärt wie Soya die ständig benötigten 3D Daten laden und auch speichern kann. Dies ist die Grundlage um Soya3D Applikationen zu entwickeln und wird insbesondere für das Kapitel Blender vorausgesetzt. Die anderen Kapitel wie Ereignisbehandlung, Kollisionsbehandlung, Einbinden in anderen Toolkits, Sound, Netzwerk und Raypicking sind je nach Bedürfnis und Interesse für sich zu studieren.

Kursiv geschriebene Wörter werden im Kapitel **Glossar** erklärt. Dabei handelt es sich hauptsächlich um Bibliotheken und Programme rund um Soya3D.

Es gibt einen **Anhang** zu diesem Tutorial in dem alle Programme und notwendige Models und Texturen vorhanden sind. Dieser ist unter den Namen attachment.tar.gz bitte separat downzuloaden und zu extrahieren. Am Ende des Tutorials gibt es die **Listings**, welche den Zusammenhang der Codesegmente hier und den Dateien herstellen. Dieser Anhang ist nicht vollständig unser Werk, es wurden große Teile von dem offiziellen Beispielprogrammen übernommen. Die Kommentare sind deshalb in Englisch, der Konsistenz wegen auch in unseren Teilen.

Das **Literaturverzeichnis** vertieft verschiedene Aspekte rund um 3D Programmierung. Wer tiefer in Soya3D einsteigen will, dem sei die API-Dokumentation und die Mailingliste von Soya3D nahegelegt.

1.1 Lizenz

Soya3D ist lizensiert unter GNU GPL (v2 und später) und damit Opensource. Es kann somit in jedem Projekt verwendet werden, welche selber eine GPL oder kompatible Lizenz

verwendet. Davon ausgenommen sind allerdings Programme welche nicht veröffentlicht werden.

Listing 1.1: Beginn der GPL-2 Lizenz

```
GNU GENERAL PUBLIC LICENSE
    Version 2, June 1991
```

Die volle Lizenz ist im Anhang zu lesen.

Die Rechte die sich durch die GPL ergeben sind freie Benutzung und Weitergabe des Codes. Zudem darf der Code, auch der von Soya3D, nach Belieben verändert werden und dieses neue abgeleitete Werk weitergegeben werden.

1.2 Leistungsmerkmale

1.2.1 Allgemeines

Soya3D hat durch Python eine portable Lösung die zumindest auf Linux, MacOSX und einigen Windows- und Unixversionen funktioniert.

Objekte werden von Soya3D direkt verwaltet, ohne dass man sich um die Matrizen kümmern muss. Neben beleuchteten Gegenständen gibt es auch noch Festkörpervolumen. Es werden auch einige Schattierungsmöglichkeiten unterstützt, so auch *Cel Shading*. Um Reflexionen und Spiegelungen darzustellen, bietet Soya3D auch Environment Mapping an. Durch Partikelsysteme lassen sich eine große Anzahl von Objekten animieren.

Es handelt sich um ein vollständiges Framework um 3D Programme zu erstellen. Deshalb werden neben der 3D Ausgabe auch noch Interaktionen unterstützt. Diese werden durch Zuhilfenahme von *SDL* durchgeführt, es kann sowohl Maus, Tastatur als auch Joystick zur Eingabe dienen. Neben Darstellung der 3D Applikation in Fenster wird selbstverständlich auch eine Vollbildschirmdarstellung unterstützt.

Es wird zudem eine Möglichkeit geboten festzustellen, ob sich ein Strahl mit einem Objekt schneidet, auch Raypicking genannt. Eine besondere Stärke sind aufwendige und schöne Landschaften. Es ist auch möglich 3D Charaktere zu animieren. Dabei wird *Cal3D* verwendet, eine Bibliothek für Charakteranimation. Soya3D verwendet auch ein eigenes Format für 3D Modelle. Es existieren aber Skripte um zwischen *Blender*, Obj/Mtl, Maya und 3DSmax zu konvertieren.

Soya3D ist in *Pyrex* geschrieben. Pyrex[13] wurde speziell dafür entwickelt um Python-Module zu schreiben und erlaubt es auf einfache Weise C-Funktionen aufzurufen.

1.2.2 Soya3D - Spezifisch

Diese hier beschriebene Funktionalität findet man in anderen 3D Engines oftmals nicht.

Sehr zur Vereinfachung und schneller Implementierung tragt das automatische Behandeln der Koordinatensystemen bei. Die Renderqualität und -geschwindigkeit wird automatisch geregelt. Bei statischen Objekten wird zu häufiges Neuberechnen oftmals verhindert. Alle Objekte, auch welche die mit Subtyping selbst abgeleitet werden, können gespeichert werden ohne weiteren Code dafür zu schreiben. Die dabei verwendete

Technik ist *Serialisierung* mit den bekannten Techniken Pickle oder Cerealizer von Python.

1.3 Ziele

- Einfaches und rasches Entwickeln von 3D Applikationen und Spielen. Soya3D zielt auf Amateurentwickler ab, welche ihre Programme in ihrer kurzen Freizeit schreiben.
- Einfach und schnell erlernbar; es soll auch für Personen ohne 3D Hintergrundwissen möglich sein damit umzugehen.
- Geschwindigkeit und Performance dürfen dabei aber nicht geopfert werden.
- Die API ist nicht angepasst auf mathematische oder technische Realität. Soya übernimmt die gesamte Arbeit der Matrizen.
- Python Eigenheiten sollen immer wenn möglich berücksichtigt werden. Objekt Serialisierung und Python Module sollen einem Python Programmierer gewohnt vorkommen und Soya3D zielt nicht darauf ab, in anderen Programmiersprachen verwendet werden zu können.
- Mache neue Bibliotheksabhängigkeiten wenn es sinnvoll ist. Jedes gute Betriebssystem hat eine Paketverwaltung welches einfache Installation vieler Abhängigkeiten gewährleistet.
- Defaultparameter repräsentieren den häufigsten Benutzungsfall.
- Soya3D versucht nicht eine typische 3D Engine zu entwickeln. Soya3D ist mehr ein Forschungsprojekt, welches versucht neue einfache Möglichkeiten für 3D aufzudecken. Es wird nicht versucht, jeden zufrieden zu stellen, soll aber ideal für eine bestimmte Gruppe sein.

2 Grundlagen

2.1 Das erste Programm

Um diese Programme hier ausprobieren zu können, muss wie zuvor erwähnt zuerst ein Paket, welches mit diesem Buch mitgeliefert wird, installiert werden. Darin enthalten sind neben den Listings auch die Daten der Modelle. In diesem sogenannten data path liegen einige Subdirectories, welche im Kapitel 5.1 noch ausführlich erklärt werden. Obwohl alle Listings vorhanden sind, wird durchaus empfohlen, sie nochmals abzutippen und dann durch Änderungen der Zeilen und Studium dieses Tutorials ein tieferes Verständnis zu erlangen.

Um Soya zu initialisieren, genügen 3 Zeilen.

```
import soya
```

Die erste Zeile kümmert sich um das Laden der Module. Die wichtigste Funktionalität ist direkt im Modul `soya` enthalten, es gibt aber auch Sub-Pakete die bei Bedarf eingeführt werden.

```
soya.path.append("/your/data/path")
```

Der Pfad muss auf den Ordner `data/` in dem zuvor erwähnten extrahierten Verzeichnis zeigen. Liegt das Programm direkt im Ordner Tutorial, so reicht

```
soya.path.append("data")

soya.init("My_3D_app")
```

Die letzte Zeile zur Initialisierung erzeugt und zeigt das 3D Display.

Jedes Soya3D Programm wird abgeschlossen mit

```
soya.MainLoop(scene).main_loop()
```

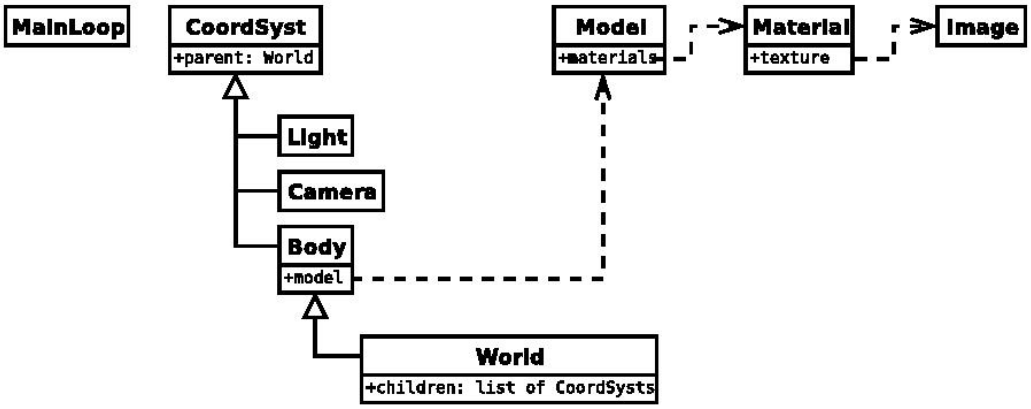
Die Hauptschleife dispatched alle Ereignisse von Soya3D - sie ist im besonderen für den Zeitverlauf zuständig.

Um ein Soya3D Programm zu beenden, ist ein

```
soya.MAIN_LOOP.stop()
```

notwendig. Dies ist noch kein vollständiges Programm, da noch keine Szene erzeugt wurde. Zuvor schauen wir uns aber noch die Klassenhierarchie von Soya3D an.

2.2 Klassenhierarchie



Der ausgefallene objektorientierte Entwurf ermöglicht die besondere Flexibilität, die Soya bietet.

Das Koordinatensystem `CoordSyst` ist die Basisklasse aller 3D Objekte. Sie beinhaltet die notwendigen Matrizen für Berechnungen, eine Orientierung und Größe. Matrizen werden aber nur in Ausnahmesituationen oder bei Erweiterungen von Soya3D direkt benötigt.

Das Licht ist somit auch ein Koordinatensystem, welches für sich alleine, wie die Kamera noch keine Aufgabe erfüllen kann. Erst dadurch, dass `World` eine Liste von `CoordSyst` aufnehmen kann, kann eine Szene geschaffen werden. In ihr beleuchtet jedes Licht die umliegenden Objekte. Die Kamera hingegen entscheidet von wo aus diese Szene gesehen und gehört wird.

Da die Welt auch indirekt von der `CoordSyst` Klasse abgeleitet wurde, ist es möglich andere Welten in ihr zu haben. Dadurch erreicht man eine Gruppierung wie in anderen 3D Engines. Diese Einheit hat vor allem den Vorteil dass durch Bewegung von `World` alle in ihr vorhanden `CoordSysts` sich auch mitbewegen.

Dadurch kann eine baumartige Struktur von `World` mit `CoordSyst` an den Blättern aufgebaut werden.

2.3 Die erste Szene

Unsere erste Szene wird nur ein Modell, nämlich ein Schwert, darstellen. Dafür sind auch nicht viele Schritte notwendig. Es wird nur eine `World scene` erschaffen, welche den `Body sword` sowie das Licht und die Kamera beinhaltet.

Listing 2.1: [basic-1.py]Die erste Szene

```
scene = soya.World()
sword_model = soya.Model.get("sword")
```

Schauen wir uns das Schritt für Schritt an. Nachdem eine Referenz auf die Welt angefordert wird, kann bereits ein Schwertmodel geladen werden. Dieses wird normalerweise

nicht erst erschaffen, sondern aus einer Datei geladen. Dabei werden automatisch die Daten und Materialen von den richtigen Verzeichnissen geladen. Die Daten müssen dabei in einer serialisierten Form speziell für Soya vorliegen. Diese Thematik wird wie angekündigt im Kapitel 5.1 behandelt.

```
sword = soya.Body(scene, sword_model)
sword.x = 1.0
sword.rotate_y(90.0)
```

Das Modell kann aber nicht direkt in die Welt eingeblendet werden. Zuvor muss ein Konstruktor `Body` ausgeführt werden, welcher die Parameter `scene` und `sword_model` entgegennimmt. Damit wird bereits ein neuer `Body sword` erschaffen und zu der Szene hinzufügt.

Die Attribute x,y,z werden am Beginn mit 0 initialisiert. Damit ist das stumpfe Ende des Schwertes genau in der Mitte des Bildschirmes platziert. Deshalb wird das Attribut x auf 1 gesetzt, was bewirkt dass es an den rechten Rand geschoben wird.

Allerdings würde das Schwert noch von uns weg zeigen und wir würden nur den Knauf in der Perspektive sehen. Deshalb führen wir eine Rotation um 90 Grad um die y Achse durch um das Schwert in voller Pracht erblicken zu können. Wir halten fest, dass Soya immer Grad als Einheit bei Winkelmaßen verwendet.

```
light = soya.Light(scene)
light.set_xyz(0.5, 0.0, 2.0)
```

Hier führen wir die Beleuchtung dieser Szene ein. Wie bereits zuvor wird die Szene direkt als Parameter an den Konstruktor des Lichtes übergeben. Dadurch wird wie zuvor ein CoordSyst im Ursprung, d.h. an der Position (0,0,0) zu der Szene hinzugefügt, diesesmal aber keinen `Body` sondern ein `Light`.

Die Koordinaten sind immer relativ zu der Welt, mit den 3 Koordinaten alleine könnte keine dreidimensionale Position bestimmt werden. Es ist somit immer zu beachten, in welcher Welt neue Objekte eingebettet werden.

Durch die Methode `set_xyz` können gleich alle 3 Attribute verändert werden. Der erste Parameter gibt dabei die Verschiebung nach rechts an, der zweite verschiebt nach oben und der letzte bringt das Licht näher an die Kamera.

```
camera = soya.Camera(scene)
camera.z = 2.0
```

Dazu müssen wir die Kamera aber erst einmal überhaupt platzieren. Das Objekt wird in der Factory (das ist ein Entwurfsmuster) wie erwartet mit `Camera` und der Szene als Parameter erzeugt und liefert uns ein Auge am Ursprung. Da wir das Schwert und das Licht bereits so ausgerichtet haben, dass wir von einiger positiver Entfernung in die negative z-Achse zum Ursprung schauen wollen, bewegen wir die Kamera um zwei Einheiten in besagte Richtung.

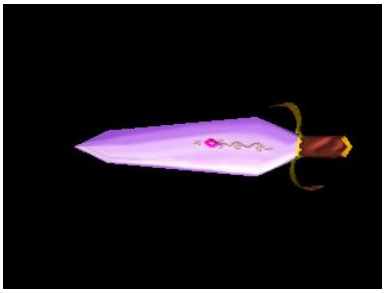
Damit haben wir ein Koordinatensystem, indem alle weiteren CoordSys am Ursprung platziert werden. Positive z Werte bewegen die Objekte zu der bzw. hinter die nicht rotierte Kamera. Die x-Achse erstreckt sich von negativen Werten links bis zu positiven Werten rechts. Und schließlich die y-Achse hat ihre negativen Werte unten. Dabei ist immer zu beachten, dass sich diese Begebenheit mit Rotationen verändern.

Eine alternative und sehr bequeme Herangehensweise für die Kamera ist die Methode `look_at` welche zu einem Vektor, Punkt oder CoordSyst hinschaut.

```
soya.set_root_widget(camera)
soya.MainLoop(scene).main_loop()
```

Das Programm schließt damit ab, dass die Kamera mit `set_root_widget` deklariert wird. Erst damit wird auch etwas in dem Fenster angezeigt. Die letzte Zeile ist notwendig um Ereignisse weiterzuleiten und das Fenster nicht sofort wieder verschwinden zu lassen.

Damit erhalten wir folgendes Ergebnis:



In diesem Beispiel ist es nicht vorgesehen, dass das Programm beendet werden kann. So wird auch ignoriert, wenn man das Fenster schließen will, nur durch ein Signal (z.b. durch Strg+C in der Konsole unter Linux) wird das Programm beendet.

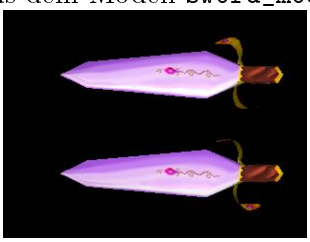
2.3.1 Mehrere Bodies

Wird das selbe Modell mehrmals mit `Model.get` geöffnet, so wird es trotzdem nur einmal geladen und mehrere Referenzen darauf zurückgegeben. Das bedeutet dass auch verschiedene Bodies mit dem gleichem Modell in einer Welt existieren können. Dies wird nachfolgend gezeigt.

Listing 2.2: [basic-1-multiple.py]Mehrere Schwerter

```
sword1 = soya.Body(scene, sword_model)
sword1.set_xyz(1.0, 0.5, 0.0)
sword1.rotate_y(90.0)

sword2 = soya.Body(scene, sword_model)
sword2.set_xyz(1.0, -0.5, 0.0)
sword2.rotate_y(90.0)
```

Dieses Listing ersetzt den entsprechenden Teil des vorigen und erzeugt nun zwei Bodies aus dem Modell `sword_model`.

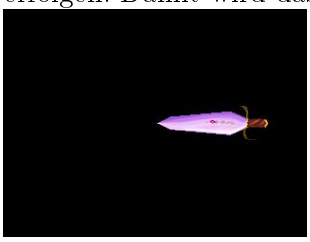


2.3.2 Skalieren

Nachdem wir Objekte jetzt schon verschoben und rotiert haben, wollen wir uns dem Skalieren widmen. Das kann mit

```
sword.scale(0.5, 0.5, 0.5)
```

erfolgen. Damit wird das erste Beispiel folgendermaßen verändert.



Dabei können aber durchaus auch die Attribute `scale_x`, `scale_y` oder `scale_z` verwendet werden. Diese Zeilen sind somit äquivalent:

```
sword.scale_x = 0.5
sword.scale_y = 0.5
sword.scale_z = 0.5
```

Jedes Koordinatensystem hat somit nicht nur eine Referenz zu einer Szene, 3 Koordinatenpositionen und Rotationen, sondern auch 3 Skalierungswerte.

2.3.3 Zeitmanagement

Die Zeit ist in Soya in 30ms Stücke geteilt. Jedes einzelne Stück mag ein wenig abweichen, aber über die Zeit gesehen wird dieser Wert angenähert.

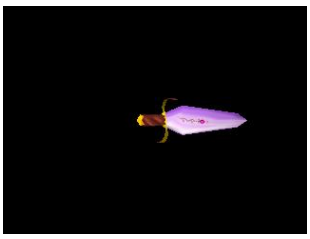
Von jedem CoordSyst werden einige Methoden automatisch innerhalb dieser Zeitspanne aufgerufen.

Rotierendes Schwert

Um das verwenden zu können, müssen wir allerdings die erste eigene Klasse erzeugen:

Listing 2.3: [basic-2.py]Rotierendes Schwert

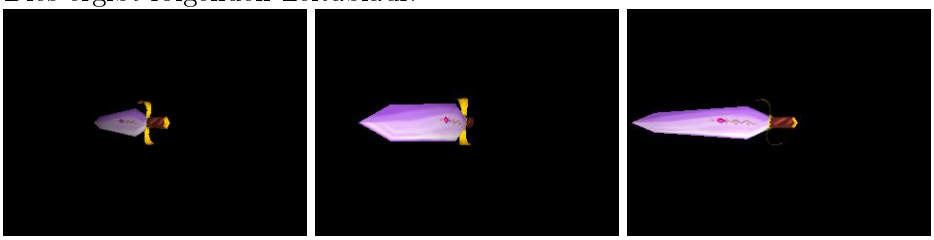
```
class RotatingBody(soya.Body):
    def advance_time(self, proportion):
        soya.Body.advance_time(self, proportion)
        self.rotate_y(proportion * 5.0)
```



In ihr wird ständig die `advance_time` Methode aufgerufen. Dies passiert für alle Objekte in der Szene. Dies passiert alle 30ms mindestens einmal. Das Argument `proportion`

zeigt an, wieviel von der Runde bereits vergangen sind. Bei 0.5 wären das 50% und somit 15ms. Mit diesem Faktor muss nun multipliziert werden um die ungleichmäßige Aufrufe zu kompensieren.

Bei jedem Timeslice rotiert das Schwert also genau 5 Grad, aber das nicht auf einmal. Dies ergibt folgenden Zeitablauf:

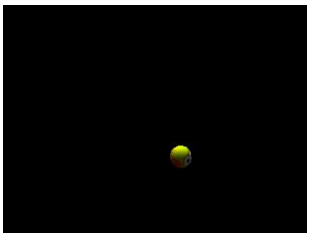


Diese Methoden, die ständig aufgerufen werden, sollten sehr minimal, hauptsächlich für Animationen, gehalten werden. Code der Entscheidungen trifft, ist woanders besser aufgehoben (siehe nächstes Kapitel).

```
sword = RotatingBody(scene, sword_model)
sword.x=-1
```

Aber zuvor wollen wir noch kurz besprechen, wie aus dieser Klasse ein rotierender Körper erzeugt wird. Dazu wird einfach der Konstruktor `RotatingBody` ausgeführt und die Referenz wie gehabt um eine Einheit in x-Richtung verschoben. Das restliche Programm ist auch ähnlich wie zuvor aufgebaut, zur Wiederholung ist es hier angegeben:

```
light = soya.Light(scene)
light.set_xyz(0.5, 0.0, 2.0)
camera = soya.Camera(scene)
camera.z = 3.0
soya.set_root_widget(camera)
soya.MainLoop(scene).main_loop()
```



Kopf der Schlange

Nun wollen wir eine Schlange programmieren. Wir fangen mit dem Kopf an und schreiben uns wieder eine neue Klasse:

Listing 2.4: [basic-3.py]Kopf der Schlange

```
class Head(soya.Body):
    def __init__(self, parent):
        soya.Body.__init__(self, parent, soya.Model.get("caterpillar_head"))
        self.speed = soya.Vector(self, 0.0, 0.0, -0.2)
        self.rotation_speed = 0.0
```

Der Konstruktor kümmert sich neben der Initialisierung von Bewegungs- und Rotationsgeschwindigkeit auch um die Initialisierung der Basisklasse Body. Hier sehen wir auch wie ein Vektor in Soya geschrieben wird, mehr dazu in Kapitel 4.2. Der erste Parameter gibt die Zugehörigkeit an.

```
    def begin_round(self):
        soya.Body.begin_round(self)
        self.rotation_speed = random.uniform(-25.0, 25.0)
```

Diesen Code wollen wir in jeder Runde garantiert nur einmal ausgeführt haben. Dies sichern wir dadurch zu, dass die Methode `begin_round` verwendet wird. Hier wird die Entscheidung getroffen, in welche Richung er sich für diese Runde bewegen soll, in diesem Fall eine Zufallszahl.

Dazu wird der Zufallsgenerator `random` von Python verwendet, welcher eine Zahl zwischen -25 und +25 zurückgibt, wie immer als Grade interpretiert.

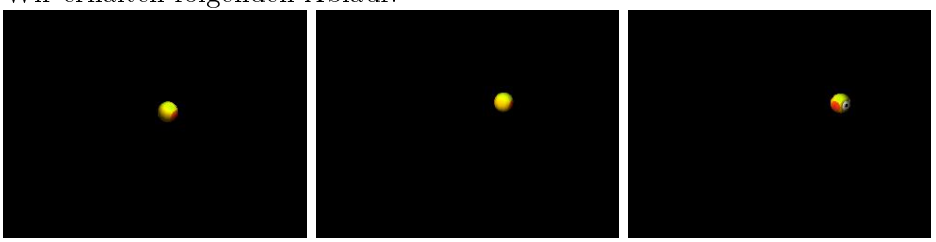
```
    def advance_time(self, proportion):
        soya.Body.advance_time(self, proportion)
        self.rotate_y(proportion * self.rotation_speed)
        self.add_mul_vector(proportion, self.speed)
```

Da jetzt für diese Runde bereits alles festgelegt ist, können wir auch ganz einfach entscheiden was in diesem Timeslice des weiteren passiert. Es wird keine Berechnung vorgenommen, sondern nur entsprechend weiter gedreht und um diese Geschwindigkeit fortbewegt.

Wichtig ist dabei, dass nicht vergessen wird jeweils mit dem Faktor `proportion` multipliziert wird. Zudem sollte man nicht vergessen den Konstruktor der Oberklasse (hier `Body`, kann aber auch z.b. `World` sein) aufzurufen.

`add_mul_vector` ist dabei nur eine beschleunigte Variante von `add_vector(proportion*self.speed)`. Sie addiert die im Konstruktor festgelegte Geschwindigkeit zu dem Vektor hinzu und multipliziert auch gleich mit `proportion`.

Wir erhalten folgenden Ablauf:



Ganze Schlange

Nun wollen wir nicht nur einen Kopf, sondern eine ganze Schlange erschaffen:

Listing 2.5: [basic-4.py]Ganze Schlange

```
class CaterpillarHead(soya.Body):
    def __init__(self, parent):
        soya.Body.__init__(self, parent, soya.Model.get("caterpillar_head"))
        self.speed = soya.Vector(self, 0.0, 0.0, -0.2)
```

```
    def begin_round(self):
        soya.Body.begin_round(self)
        self.rotate_y((random.random() - 0.5) * 50.0)
    def advance_time(self, proportion):
        soya.Body.advance_time(self, proportion)
        self.add_mul_vector(proportion, self.speed)
```

Der Kopf hat nicht viele Unterschiede zu dem vorigen Beispiel. Nur die Zufallsberechnung ist anderes und führt ausserdem direkt zu einer Rotation.

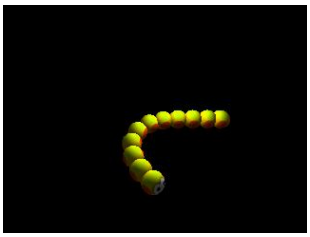
```
class CaterpillarPiece(soya.Body):
    def __init__(self, parent, previous):
        soya.Body.__init__(self, parent, soya.Model.get("caterpillar"))
        self.previous = previous
        self.speed = soya.Vector(self, 0.0, 0.0, -0.2)
    def begin_round(self):
        soya.Body.begin_round(self)
        self.look_at(self.previous)
        if self.distance_to(self.previous) < 1.5: self.speed.z =  0.0
        else:                                     self.speed.z = -0.2
    def advance_time(self, proportion):
        soya.Body.advance_time(self, proportion)
        self.add_mul_vector(proportion, self.speed)
```

Die Körperelemente haben aber ein anderes Ziel, sie folgen dem Teil davor. Dies ist mit der bereits eingeführten Methode `look_at` sehr einfach realisiert. Dadurch muss nur `speed.z` angepasst werden, je nachdem ob aufgeholt werden muss oder der Teil schon knapp genug am vorigen ist.

Bei Voranschreiten der Zeit wird wieder genau das gleiche gemacht. Wir sehen dass bewegte Teile nur in der Entscheidungslogik Unterschiede haben, welche wie bereits erwähnt in `begin_round` stattfinden soll.

```
caterpillar_head = CaterpillarHead(scene)
caterpillar_head.rotate_y(90.0)
previous_caterpillar_piece = caterpillar_head
for i in range(10):
    previous_caterpillar_piece = CaterpillarPiece(scene, previous_caterpillar_piece)
    previous_caterpillar_piece.x = i + 1
```

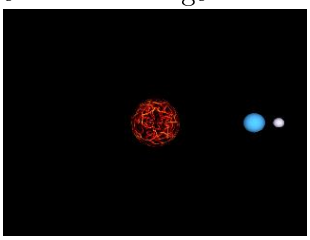
Hier wird nun der Kopf der Szene hinzugefügt und gleich gedreht damit er orthogonal zum Betrachter schaut. Nun werden in der Schleife ein Teil nach dem anderen hinzugefügt, gleich mit einem Abstand von 1. Dabei wird jeweils der vorige Teil als Referenz übergeben.



2.4 Welten in Welten

Wie wir bereits festgestellt haben, können in einer Welt auch eine andere Welt hinzugefügt werden. Das verwenden wir jetzt um ein Rotieren von einem Planeten um die Sonne und eines Mondes um den Planeten zu realisieren.

Die besprochenen Aktionen wie Verschieben, Rotieren und Skalieren können auch auf eine Welt angewandt werden. Diese haben dann aber implizit auf alle `CoordSyst` in dieser Welt Auswirkungen.



```
World---Camera
      |-Sun
      |  |-earth
      |      |-moon
      |-Light
```

Listing 2.6: [nested-world-1.py]Planetensystem

```
import sys, os, os.path, soya
soya.init()
soya.path.append(os.path.join(os.path.dirname(sys.argv[0]), "data"))
```

Diese Zeilen werden wie gewohnt verwendet.

```
scene = soya.World()
class CelestialObject(soya.World):
    def advance_time(self, proportion):
        soya.World.advance_time(self, proportion)
        self.rotate_y(proportion * 2.0)
```

Wir erschaffen einen Himmelskörper, der ein Subtyp von einer `soya.World` ist, also auch alle Eigenschaften davon erfüllt. Wir überschreiben allerdings `advance_time` in der wir die Welt in der y-Achse rotieren lassen, je timeslice um 2 Grad.

```
sun = CelestialObject(scene, soya.Model.get("sun"))
```

Einen solchen Himmelskörper, die Sonne, erschaffen wir jetzt. Damit haben wir bereits einen rotierenden Körper geschaffen.

```
earth = CelestialObject(sun, soya.Model.get("earth"))
earth.x = 2.0
```

Nun wollen wir zusätzlich einen Planeten haben, der sich um die Sonne dreht. Wie wir leicht sehen können, wird dieser Körper aber nicht der globalen Szene zugeordnet,

sondern der zuvor erstellten Sonne. Zu beachten ist, dass die x-Koordinate 2 innerhalb dieses Koordinatensystems zum Tragen kommt.

Der ganze Planet rotiert nun im Abstand 2 von der Sonne entfernt, weil im Himmelskörper Sonne eine Rotation stattfindet, sie laufen also synchron, was in der Realität nicht so ist.

Der Planet selber rotiert auch, weil er ein Himmelskörper mit eigener Rotation ist. Wir haben also in diesem Stadium bereits 2 unabhängige Drehbewegungen, die aber aufgrund der hart eincodierten 2 Grad gleich schnell vonstatten gehen. Da man am Planeten nicht erkennen kann, dass er einen Spin hat, kann man das Programm leicht umschreiben, das an dieser Stelle auch eine Sonne verwendet wird. Dies sei dem Leser als Übung überlassen.

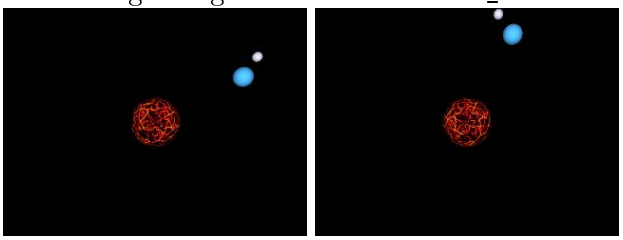
```
moon = CelestialObject(earth, soya.Model.get("moon"))
moon.x = 0.5
```

Mit nur 2 weiteren Zeilen können wir hier einen Mond in unserem Sonnensystem einfügen. Die x-Koordinate ist wie zuvor in der relativen Welt, hier des Planeten zu sehen.

Wir führen wieder eine neue Drehbewegung ein, die wir aber aufgrund der Gestalt des Mondes aber nicht wahrnehmen.

```
light = soya.Light(scene)
light.set_xyz(0.0, 5.0, 0.0)
camera = soya.Camera(scene)
camera.y = 4.0
camera.look_at(soya.Vector(scene, 0.0, -1.0, 0.0))
soya.set_root_widget(camera)
soya.MainLoop(scene).main_loop()
```

Dieser Teil ist wie gehabt, nur verwenden wir das komfortable `look_at` um auf die Szene zu schauen. Betrachtet wird alles von der Kamera aus, welche 4 Einheiten in y Achse entfernt ist. Deshalb müssen wir in die negative y-Achse schauen, was wir mit einer beliebigen negativen Zahl bei `look_at` erreichen.



```
earth = CelestialObject(None, soya.Model.get("earth"))
sun.add(earth)
```

Eine andere Möglichkeit wäre gewesen, im Konstruktor noch keine Eltern anzugeben und stattdessen das Objekt mit add hinzuzufügen. Der Vorteil davon ist, dass das Objekt dynamisch auch wieder durch folgende Zeile entfernt werden kann:

```
sun.remove(earth)
```

Es kann auch jederzeit abgefragt wer die Eltern eines Objektes sind, diese Variable kann aber nur gelesen werden.

```
if moon.parent is earth: print "OK"
```

In die andere Richtung kann man auch abfragen wer die Kinder einer Szene sind.

```
print earth in sun.children # => true
print moon in sun.children # => false
```

Diese Kinder können wie in Python üblich als Liste behandelt werden. Dadurch kann man auch mit for darüber iterieren und wie folgend alle Kinder ausgeben.

```
for coord_syst in sun:
    print "the sun contains", coord_syst
```

Bei einer Suche ist man aber nicht nur auf die direkten Kinder beschränkt. Es gibt auch rekursive Abfragen.

```
print earth.is_inside(sun) # => true
print moon .is_inside(sun) # => true
```

Dadurch kann man erfragen ob ein bestimmtes Objekt in einer beliebigen Tiefe vorhanden ist oder nicht.

Eine andere Suche funktioniert über eine anonyme Funktion. Hier kann `CoordSyst.name` sehr behilflich sein.

```
scene.search_all(lambda coord_syst: isinstance(coord_syst, soya.Body)
                                    and coord_syst.model is soya.Model.get("moon"))
```

Welten können auch zu einem Modell verwandelt werden, durch die Verwendung der Methode `to_model`. Der erhöhten Geschwindigkeit steht gegenüber, dass keine weiteren Änderungen mehr vorgenommen werden können.

2.5 Geometrische Objekte

Soya3D unterstützt vordefinierte Modelle und bietet auch die Möglichkeit welche selber zu erzeugen. Diese Vorgehensweise ist aber nur für sehr primitive oder algorithmisch leicht erzeugbare Modelle durchführbar, für komplexere ist *Blender* eine unverzichtbare Hilfe, siehe Kapitel 10.

2.5.1 Vordefinierte Modelle

Soya3D bietet lediglich 2 vordefinierte Modelle: eine Kugel und einen Würfel. Diese befinden sich in Unterpaketen von Soya3D und müssen daher zusätzlich importiert werden, wenn man sie verwenden will.

Listing 2.7: [mymodel.py]Vordefinierte Modelle

```
import soya.sphere, soya.cube
```

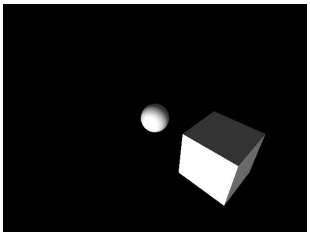
Nachdem man die zusätzlichen Pakete installiert hat und die Welt erzeugt hat (siehe 2.3), erzeugt man zuerst die Modelle.

```
m_sphere = soya.sphere.Sphere(None,None).shapify()
m_cube = soya.cube.Cube(None, None, size=3).shapify()
```

Aus diesen Referenzen kann man die konkreten Objekte erzeugen und platzieren.

```
sphere=soya.Body(scene,m_sphere)
cube=soya.Body(scene,m_cube)

cube.set_xyz(5,0,0)
```

Zuletzt fehlt nur noch Licht und Kamera, dann sieht das ganze wie folgt aus:



2.5.2 Selbstdefinierte

Wer andere einfache Objekte haben will wie z.B. eine Pyramide, kann sich diese mittels den sogenannten Faces selber erzeugen. Der Code dazu sieht so aus:

Listing 2.8: [modeling-1.py]Faces

```
pyramid = soya.World(scene)
soya.Face(pyramid, [soya.Vertex(pyramid,  0.5, -0.5,  0.5),
    soya.Vertex(pyramid, -0.5, -0.5,  0.5),
    soya.Vertex(pyramid, -0.5, -0.5, -0.5),
    soya.Vertex(pyramid,  0.5, -0.5, -0.5),
    ])
soya.Face(pyramid, [soya.Vertex(pyramid, -0.5, -0.5,  0.5),
    soya.Vertex(pyramid,  0.5, -0.5,  0.5),
    soya.Vertex(pyramid,  0.0,  0.5,  0.0),
    ])
soya.Face(pyramid, [soya.Vertex(pyramid,  0.5, -0.5, -0.5),
    soya.Vertex(pyramid, -0.5, -0.5, -0.5),
    soya.Vertex(pyramid,  0.0,  0.5,  0.0),
    ])
soya.Face(pyramid, [soya.Vertex(pyramid,  0.5, -0.5,  0.5),
    soya.Vertex(pyramid,  0.5, -0.5, -0.5),
    soya.Vertex(pyramid,  0.0,  0.5,  0.0),
    ])
soya.Face(pyramid, [soya.Vertex(pyramid, -0.5, -0.5, -0.5),
    soya.Vertex(pyramid, -0.5, -0.5,  0.5),
    soya.Vertex(pyramid,  0.0,  0.5,  0.0),
    ])
pyramid.filename = "pyramid"
pyramid.save()
```

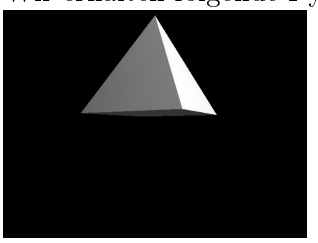
Wie wir sehen, müssen die einzelnen Eckpunkte (engl. Vertex, siehe 4.1), angegeben werden. Die Liste wird durch `soya.Face` erzeugt, wobei als erster Parameter jeweils das Objekt angegeben werden muss.

Damit wird die Pyramide als Modell gespeichert. Sie kann jetzt wie ein Modell verwendet werden:

Listing 2.9: [modeling-2.py]Modell laden

```
pyramid_model = soya.Model.get("pyramid")
pyramid1 = soya.Body(scene, pyramid_model)
pyramid1.x = -0.7
pyramid1.rotate_y(60.0)
pyramid2 = soya.Body(scene, pyramid_model)
pyramid2.x = 0.8
pyramid2.rotate_y(45.0)
```

Wir erhalten folgende Pyramide:



Bei der Erzeugung der Pyramide müssen sehr viele Eckpunkte doppelt für die Beschreibung der Kanten angegeben werden.

Fünf Punkte reichen aus um das Objekt zu beschreiben. Wir führen hier auch gleich diffuse ein, eine Möglichkeit Punkten eine Farbe zuzuordnen.

Listing 2.10: [modeling-3.py]Faces und Farben

```
pyramid_world = soya.World()
apex  = soya.Vertex(pyramid_world,  0.0,  0.5,  0.0, diffuse = (1.0, 1.0, 1.0, 1.0))
base1 = soya.Vertex(pyramid_world,  0.5, -0.5,  0.5, diffuse = (1.0, 0.0, 0.0, 1.0))
base2 = soya.Vertex(pyramid_world, -0.5, -0.5,  0.5, diffuse = (1.0, 1.0, 0.0, 1.0))
base3 = soya.Vertex(pyramid_world, -0.5, -0.5, -0.5, diffuse = (0.0, 1.0, 0.0, 1.0))
base4 = soya.Vertex(pyramid_world,  0.5, -0.5, -0.5, diffuse = (0.0, 0.0, 1.0, 1.0))
```

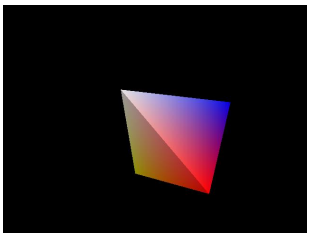
Mit diesen Punkte bauen wir jetzt die Pyramide auf:

```
soya.Face(pyramid_world, [base1, base2, base3, base4])
soya.Face(pyramid_world, [base2, base1, apex])
soya.Face(pyramid_world, [base4, base3, apex])
soya.Face(pyramid_world, [base1, base4, apex])
soya.Face(pyramid_world, [base3, base2, apex])
```

Jetzt benötigen wir nur noch eine Umwandlung der Welt zu einem Modell. Dies bringt einen Geschwindigkeitsvorteil, verunmöglicht aber späteres Ändern.

```
pyramid_model = pyramid_world.to_model()
```

Damit wird diese bunte Pyramide erzeugt:



2.6 Material

Man kann auf ein Objekt eine Textur setzen. Dies wird bei Soya3D als Material bezeichnet, wobei auch das Zuordnen einer Farbe zu einem Objekt darunter fällt.

Listing 2.11: [material.py]Material

```
texture = soya.Material(soya.Image.get("metal1.png"))

red   = soya.Material();
red.diffuse = (1.0, 0.0, 0.0, 1.0)

green = soya.Material();
green.diffuse = (0.0, 1.0, 0.0, 1.0)

blue  = soya.Material();
blue.diffuse = (0.0, 0.0, 1.0, 1.0)

transparent  = soya.Material();
transparent.diffuse = (1.0, 1.0, 1.0, 0.5)
```

Hier werden vier verschieden Materialien definiert. Zuerst wird eine Textur aus einem Bild geladen. Das Bild muss dabei im Unterverzeichnis `images` im Data-Dir (siehe) abgespeichert sein. Die nächsten drei Materialien sind die Farben Rot, Grün und Blau. Diese wurden über RGBA-Tupel (Rot, Grün, Blau, Alpha) angegeben. Jeder Farbkanal kann dabei einen Wert zwischen 0.0 und 1.0 haben. Interessant ist dabei auch der Alpha-Kanal. Über diesen kann man die Transparenz angeben. Dies wird beim letzten Material `transparent` ausgenutzt.

Mit den Materialien müssen wir jetzt die Models erzeugen. In diesem Beispiel erzeugen wir verschiedenfarbige Würfel. Beim transparenten Würfel ist noch etwas zusätzliche Arbeit notwendig, da Soya3D Backface Culling[2] verwendet. Damit werden Faces die nicht in Kamera-Richtung zeigen, nicht gezeichnet, da sie im Normalfall ohnehin durch andere überdeckt werden. Dies ist aber bei Transparenz nicht der Fall und daher müssen wir diese Optimierung deaktivieren. Dies geschieht indem wir für alle Faces das Attribut `double_sided` auf 1 setzen.

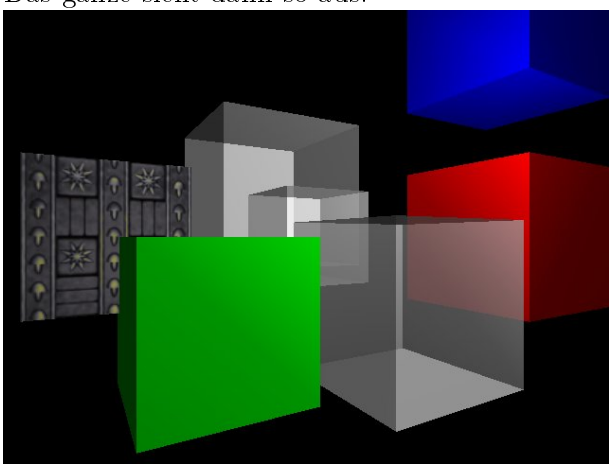
```
m_cube_texture = soya.cube.Cube(material = texture).shapify()
m_cube_red = soya.cube.Cube(material = red).shapify()
m_cube_blue = soya.cube.Cube(material = blue).shapify()
m_cube_green = soya.cube.Cube(material = green).shapify()

cube_transparent = soya.cube.Cube(material = transparent)
for face in cube_transparent.children:
    face.double_sided = 1
cube_model = cube_transparent.to_model()
```

Nachdem man die Models hat, kann man damit die Objekte erzeugen. Diese werden dann noch an die gewünschten Stellen verschoben und rotiert, damit man einen besseren räumlichen Eindruck hat.

```
cube1 = soya.Body(scene, cube_model)
cube1.set_xyz(-0.3, 0.3, 0.0)
```

```
cube1.rotate_y(30.0)

cube2 = soya.Body(scene, cube_model)
cube2.set_xyz(0.0, 0.0, -1.5)
cube2.rotate_y(30.0)

cube3 = soya.Body(scene, cube_model)
cube3.set_xyz(0.6, -0.4, 0.5)
cube3.rotate_y(30.0)

cube_texture = soya.Body(scene, m_cube_texture)
cube_texture.set_xyz(-1.5, 0, 0)
cube_texture.rotate_y(30.0)

cube_red = soya.Body(scene, m_cube_red)
cube_red.set_xyz(1.5, 0, 0)
cube_red.rotate_y(30.0)

cube_blue = soya.Body(scene, m_cube_blue)
cube_blue.set_xyz(1.5, 1.5, 0)
cube_blue.rotate_y(30.0)

cube_green = soya.Body(scene, m_cube_green)
cube_green.set_xyz(-0.6, -0.5, 0.5)
cube_green.rotate_y(30.0)
```

Das ganze sieht dann so aus:



2.7 Frames Per Second

Abschließend wollen wir noch kurz zeigen wie man Frames Per Second (FPS) anzeigen kann. Dieser Wert gibt eine Größe an ob die Hardware mit dem Bildfrequenz noch schnell genug ist, oder ob die Gefahr unerwünschter Effekte wie ruckeln sehr groß ist. Sehr hohe Zahlen bieten keinerlei Vorteil.

Damit man die Frames Per Seconds (FPS) angezeigt bekommt, fügt man dem `root_widget` ein `FPSLabel()`-Widget hinzu. Da man jedoch als root-Widget meistens die Kame-

ra benötigt, ersetzt man dies als Root-Widget durch ein Group-Widget. Diesem kann man dann mehrere andere Widget hinzufügen, in unserem Fall die Kamera und das `FPSLabel()`.

Listing 2.12: [pudding-fps.py]Anzeigen von Frames per Second

```
soya.set_root_widget(widget.Group())
soya.root_widget.add(character.camera)
soya.root_widget.add(widget.FPSLabel())
```

Die FPS werden dann rechts unten angezeigt und automatisch aktualisiert.

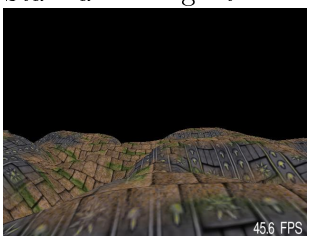
3 Grafische Effekte

Dieses Kapitel zeigt erweiterte grafische Effekte, die nicht die Funktionalität erweitern, aber das Aussehen der Bilder verändern.

3.1 Atmosphäre

In diesem Beispiel werden Terrains verwendet, genaueres darüber folgt im Kapitel 8.4 Die Atmosphäre dient dazu, das ambiente Licht zu verändern und eine Nebelwirkung zu erzielen.

Standardmäßig ist der Hintergrund schwarz:

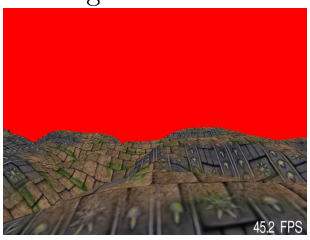


Wir können das leicht auf knallrot abändern, indem wir die Hintergrundfarbe verändern:

Listing 3.1: [atmosphere.py]Roter Hintergrund

```
scene.atmosphere = soya.Atmosphere()
scene.atmosphere.bg_color = (1.0, 0.0, 0.0, 1.0)
```

Das ergibt nun:



Nebel kann mit `fog` aktiviert werden und die Farbe mit `fog_color` verändert werden.

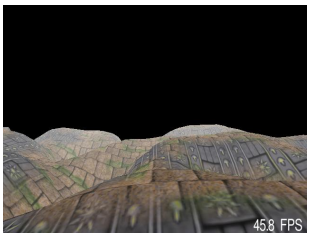
Es gibt auch einige Typen wie dicht Nebel ist. Diese können mit `fog_type` verändert werden. 0 steht für lineare Dichte, 1 exponentiell und 2 exponentiell zum Quadrat. Dabei wird bei linearer Dichte `fog_start` und `fog_end` berücksichtigt. Bei den exponentiellen Modellen hingegen kann die Gesamtdichte mit `fog_density` verändert werden.

Listing 3.2: [atmosphere.py]Linearer Nebel

```
scene.atmosphere = soya.Atmosphere()
```

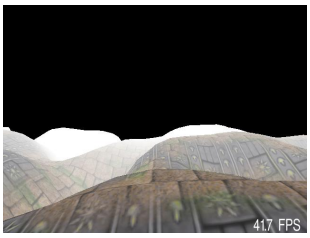
```
scene.atmosphere.ambient = (1.0, 1.0, 1.0, 1.0)
scene.atmosphere.fog = 1
scene.atmosphere.fog_type = 0
scene.atmosphere.fog_start = 0.0
scene.atmosphere.fog_end = 50.0
scene.atmosphere.fog_color = (1.0, 1.0, 1.0, 1.0)
```

Beginnen wir einmal mit einem linearen weißen Nebel der bei 50 Entfernung aufhört, also die gewählte Farbe, hier weiß, annimmt.

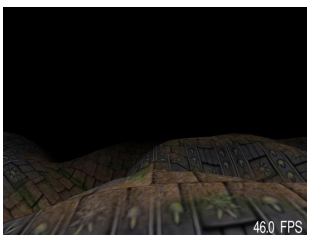



```
scene.atmosphere = soya.Atmosphere()
scene.atmosphere.fog_end = 20.0
scene.atmosphere.fog_color = (1.0, 1.0, 1.0, 1.0)
```

Nun wollen wir die Distanz auf 20 verringern:



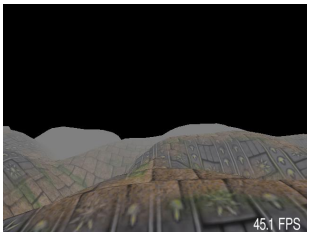
Probieren wir das ganze einmal mit der Defaultfarbe Schwarz:



Zum Schluss testen wir noch einen Grauwert, der ein hübscheres Ergebnis als der weiße liefert.

Listing 3.3: [atmosphere.py]Grauer Nebel

```
scene.atmosphere = soya.Atmosphere()
scene.atmosphere.ambient = (1.0, 1.0, 1.0, 1.0)
scene.atmosphere.fog_end = 20.0
scene.atmosphere.fog_color = (0.5, 0.5, 0.5, 1.0)
```



3.2 Schattierung

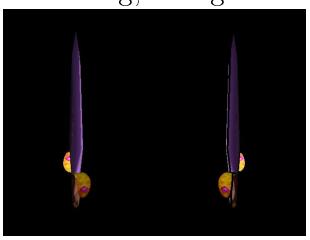
Schattierung oder auch Shading genannt wird durch ein Beleuchtungsmodell berechnet und verändert die Oberfläche in Abhängigkeit von dem Schattierungsmodell und der Beleuchtung. Soya3D unterstützt hier zwei Modelle.

Standardmäßig wird mit Opengl die Schattierung berechnet. Für jede Welt kann aber eine andere Schattierung verwendet werden. Eine zweite Schattierung, die Soya3D unterstützt nennt sich *Cel Shading*. Cel steht für den englischen Ausdruck "celluloid". Leider hat sich bei Soya3D ein kleiner Fehler eingeschlichen, und die Klasse nennt sich "Cell Shading".

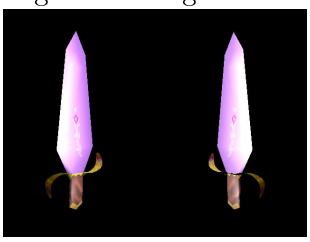
Listing 3.4: [shading.py]Shading

```
sword = soya.World.get("sword")
material = sword.children[0].material
material.texture = soya.Image.get("epee_turyle-cs.png")
material.separate_specular = 1
material.shininess = 15.0
material.specular = (1.0, 1.0, 1.0, 1.0)
```

Lassen wir die letzten 3 Zeilen weg, so ergeben sich folgende Bilder:

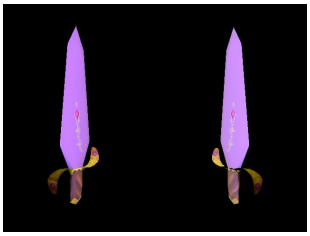


Setzt man hingegen den Glanz (shininess) auf 15 und aktiviert ein weißes Glanzlicht, so ergibt sich folgendes:



Selbstverständlich ist auch ein blaues Glanzlicht möglich:

```
material.specular = (0.0, 0.0, 1.0, 1.0)
```



```
sword_model = sword.to_model()
shader = soya.Material()
shader.texture = soya.Image.get("shader.png")
```

Hier wird zuerst der Shader initialisiert. Wie wir sehen ist er durch eine Textur realisiert. In der folgenden Zeile müssen wir auch den richtigen ModelBuilder für Cel Shading verwenden, da defaultmäßig kein Cel Shading durchgeführt wird.

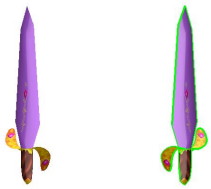
```
celshading = soya.CellShadingModelBuilder()
celshading.shader                = shader
celshading.outline_color         = (0.0, 0.0, 0.0, 1.0)
celshading.outline_width         = 7.0
celshading.outline_attenuation = 1.0
sword.model_builder = celshading
sword_celshaded_model = sword.to_model()
```

Die `outline_color` spezifiziert die Umrißlinie. Sie ist defaultmäßig schwarz und wird hier absichtlich auch auf Schwarz gesetzt.

Grün kann folgendermaßen verwendet werden:

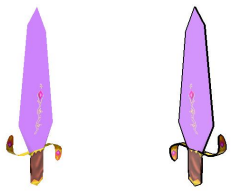
```
celshading.outline_color         = (0.0, 1.0, 0.0, 1.0)
```

Dadurch ergibt sich folgendes Bild:



Die Dicke (width) ändert die der Umrißlinie.

Die Abschwächung (attenuation) spezifiziert wie der Abstand die Dicke des Umrißes ändert. Abschließend noch ein Bild mit den Defaultwerten:



3.3 Schatten

Schattenwurf ist ein essentieller Punkt bei realistischen Darstellungen. Erst dadurch wird eine Tiefe vermittelt und Objekte beeinflussen sich bei Beleuchtung gegenseitig.

Listing 3.5: [modeling-shadow-1.py]Schatten

```
scene = soya.World()
sword = soya.World.get("sword")
model_builder = soya.SimpleModelBuilder()
model_builder.shadow = 1
```

Schatten werden eingeschalten durch direkte Beeinflussung des ModelBuilder mit dem wir schon zu tun hatten. Um sie zu aktivieren muss das Attribut `shadow` auf 1 gesetzt werden.

```
sword.model_builder = model_builder
sword_model = sword.to_model()
```

Anschließend wird für das Schwert der ModelBuilder verwendet, wo Schatten aktiviert sind. Dieses Objekt wird mit `to_model()` zu einem Modell transformiert.

```
wall_model = soya.World()
wall_face = soya.Face(wall_model, [
    soya.Vertex(wall_model, 0.0, -5.0, -5.0),
    soya.Vertex(wall_model, 0.0, -5.0,  5.0),
    soya.Vertex(wall_model, 0.0,  5.0,  5.0),
    soya.Vertex(wall_model, 0.0,  5.0, -5.0),
    ])
```

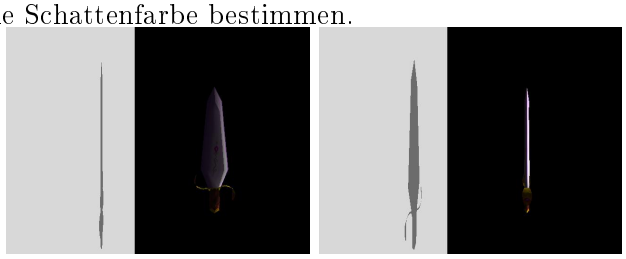
Wir erzeugen mit einem Face eine Wand, auf die der Schatten projiziert werden soll.

```
wall_face.double_sided = 1
wall = soya.Body(scene, wall_model.to_model())
wall.set_xyz(-1.0, 0.0, 0.0)
```

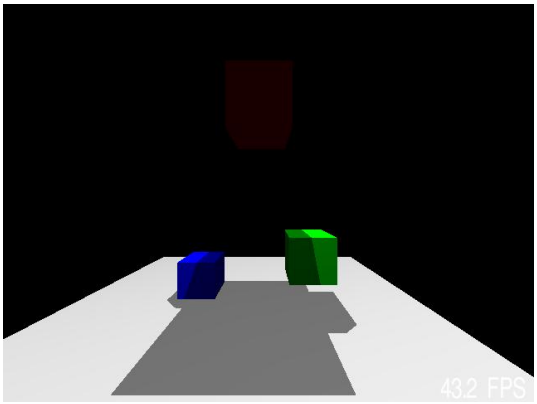
Da das Face nur eine Ebene ist, und damit durchsichtig, muss das Attribut `double_sided` gesetzt werden. Jetzt ist auf beiden Seiten eine Vorderseite und der Schatten kann darauf geworfen werden.

```
light = soya.Light(scene)
light.set_xyz(5.0, 0.0, 0.0)
light.shadow_color = (0.0, 0.0, 0.0, 0.5)
light.cast_shadow = 1
```

Bei dem Licht muss man `cast_shadow` angeben. Zudem kann man mit `shadow_color` die Schattenfarbe bestimmen.



Hier noch ein Bild von mehreren Objekten, die sich durch Schatten beeinflussen:



Schatten werden aber auch von animierten Objekten richtig geworfen:

3.4 Spuren

Spuren (in Soya Rays) ist ein grafischer Eindruck von Bewegung auf Objekten. Dabei bleibt eine gewisse Resthelligkeit bestehen wo ein Objekt, welches diese Eigenschaft zugewiesen bekommt, gewesen ist.

```
material.shininess = 0.5
material.diffuse    = (0.6, 0.5, 0.7, 1.0)
material.additive_blending = 1
```

Wir ändern die Materialeigenschaften um einen schöneren Effekt zu erhalten. Besonders eindrucksvoll ist `additive_blending` welches zusätzlich noch den Bewegungseindruck verstärkt.

```
ray.z = -0.2
ray.endpoint = soya.Point(sword, 0.0, 0.0, -1.95)
ray.material = material
```

Nun erzeugen wir das Ray Objekt für das Schwert. Als Länge wird mit `length` 20 übergeben, default ist 10.

3.5 Environment Mapping

Environment Mapping[2] ist eine Technik, um spiegelnde Oberflächen zu realisieren. Dabei wird eine Textur verwendet um die Umgebung, die reflektiert werden soll, darzustellen. Die dabei verwendete Textur wird "sphere map" genannt und kann zum Beispiel in Gimp [15] mit dem Polar Koordinaten Effekt erzeugt werden.

Der Vorteil von Environment Mapping im Gegensatz zu anderen Techniken, wie z.B. Ray Tracing ist, dass es sehr schnell ist – der Nachteil jedoch, dass die Darstellung bestimmte Näherungen voraussetzt. So wird angenommen, dass das Objekt inmitten der Szene ist und andere Objekte in der Nähe nicht gespiegelt werden. Das ist allerdings besonders bei Animationen nicht leicht zu erkennen.

Im nächsten Beispiel verwenden wir folgende Textur für die Umgebung:

Diese Textur wird nun als Material für Environment Mapping geladen.

Dies geschieht wie das Laden einer gewöhnlichen Textur, nur dass zusätzlich das At-

tribut `environment_mapping` auf `1` gesetzt wird.

Listing 3.8: [modeling-env-mapping-1.py]Environment Mapping

```
material = soya.Material()
material.environment_mapping = 1 # Specifies environment mapping is active
material.texture = soya.Image.get("sphere_map.jpg")# The textured sphere map
```

Dann erzeugen wir die Welten, die sich einfach zu Models transformieren lassen. Dabei werden ein Würfel und zwei Kugeln erzeugt. Für jedes Face wird das Attribut `material` auf das zuvor erzeugte Material gesetzt um damit die Textur projezieren. Zusätzlich wird das Attribut `smooth_lit` gesetzt, um auszuwählen ob man die Kanten der Faces sehen soll oder nicht. Bei Objekten wie einem Würfel hat es zwar keinen Effekt, aus Performancegründen schaltet man es aber lieber ab. Bei den Kugeln setzen wir es einmal auf `1` und einmal auf `0`. Damit erreichen wir einmal eine glatte Oberfläche der Kugel und einmal eine, bei der man noch die Faces erkennen kann.

```
ball_world=soya.sphere.Sphere()
faceted_ball_world = soya.sphere.Sphere()
cube_world=soya.cube.Cube()
for face in faceted_ball_world.children:
    face.smooth_lit=0
    face.material=material
for face in ball_world.children:
    face.smooth_lit=1
    face.material=material
for face in cube_world.children:
    face.smooth_lit=0
    face.material=material
```

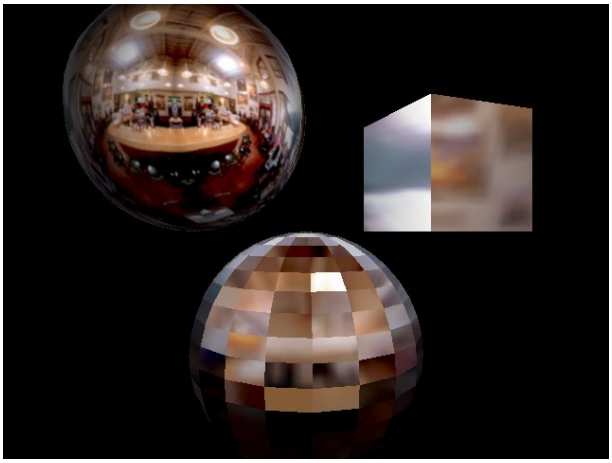
Weiters wollen wir die Objekte rotieren lassen. Dazu verwenden wir statt einem `soya.Body`, die selbstgeschriebe und abgeleitete Klasse `RotatingBody`:

```
class RotatingBody(soya.Body):
    def advance_time(self, proportion):
        self.rotate_y(2.0 * proportion)
```

Nun können wir die konkreten Objekte erzeugen und ihre Position setzen:

```
ball          = RotatingBody(scene, ball_world          .to_model())
cube          = RotatingBody(scene, cube_world          .to_model())
faceted_ball = RotatingBody(scene, faceted_ball_world.to_model())
ball.x          = -1
ball.y          =  1
cube.x          =  1.2
cube.y          =  0.5
faceted_ball.y = -1
```

Das Ergebnis sieht dann so aus:



4 Die Mathematik von Soya

4.1 Punkt

Wir wissen bereits, dass jede dreidimensionale Position durch die drei Koordinaten x, y und z sowie durch das CoordSyst in dem sie sich befindet, repräsentiert wird. Das kann man sich so vorstellen, dass insgesamt 6 Koordinaten vorhanden sind. Drei davon stellen den Punkt in einem Referenzsystem dar. Die anderen 3 hingegen legen fest wo das Referenzsystem ist. Die Darstellung des Referenzsystemes ist von der Kameraposition abhängig.

Mit **Punkten** kann Soya3D automatisch, wenn notwendig, in die Koordinaten eines anderen CoordSyst umrechnen. Dabei ändern sich natürlich die x,y und z Werte, aber auch die 3 Werte, auf die man sich bezieht, nämlich die einer anderen `World`.

```
soya.Point(CoordSyst, x, y, z)
```

Der Konstruktor erzeugt eine dreidimensionale Position, festgelegt durch das `CoordSyst`, x, y und z. Wir nennen diese Datenstruktur `Point`.

```
soya.Point(earth, 1.0, 0.0, 0.0)
```

In diesem Beispiel wird ein Punkt mit (1,0,0) in dem Referenzsystem `earth` kreiert. Zu beachten ist allerdings, dass ein Punkt nicht gesehen werden kann, er wird in der Szene nicht gerendert.

```
world = soya.World()
point = soya.Point(world, 1.0, 0.0, 0.0)
print world.children # => [] (empty list)
```

Wir sehen, dass der Punkt auch nicht als Kind gelistet ist. Punkte können in jedem `CoordSyst` auftreten.

```
print moon.distance_to(sun)
```

Die Distanz zwischen zwei `CoordSyst` oder Punkten kann durch die Methode `distance_to` bestimmt werden. Das funktioniert im Besonderen auch, wenn die zwei Objekte nicht im gleichen `CoordSyst` definiert sind.

Punkte besitzen zudem noch alle Eigenschaften von ihrem `CoordSyst` bezüglich Bewegung – von Skalierung und Rotation bleiben sie allerdings unbeeinflusst.

```
moon_center = soya.Point(moon, 0.0, 0.0, 0.0)
moon_center.convert_to(sun)
print "In the sun coordinate system, the center of the moon is", moon_center
```

Um manuell zwischen verschiedenen Koordinatensystemen zu konvertieren, kann `convert_to` verwendet werden.

```
print "In the sun coordinate system, the center of the moon is", moon % sun
```

In Python können auch Operatoren überladen werden. In diesem Zusammenhang wurde dem % (Prozent) eine neue Bedeutung zugeschrieben. Mit ihm kann man auch die Koordinaten konvertieren, allerdings nicht in-place, es wird ein neues Objekt dafür kreiert. Somit sind die letzten zwei Beispiele äquivalent.

4.2 Vektor

Für Vorwärts- und Drehbewegungen kann ein Vektor verwendet werden. `Vector` ist zwar von `Point` abgeleitet, bildet aber keine Untertypbeziehung, sondern das hat nur interne Gründe zur Codewiederverwendung.

```
soya.Vector(CoordSyst, x, y, z)
```

Der Konstruktor hat die gleichen Parameter wie `Point`, er nimmt ein `CoordSyst` und 3 Koordinaten entgegen. Der Vektor zeigt vom Ursprung zu den angegebenen (x,y,z) Koordinaten.

```
vector = CoordSyst_or_Point.vector_to(CoordSyst_or_Point)
```

die andere Möglichkeit ist die Methode `vector_to`. Mit ihr kann von einer bestimmten Position, gegeben durch ein `CoordSyst` oder `Point`, zu einer anderen, ebenfalls so gegeben, ein Vektor gebildet werden.

```
vector = CoordSyst_or_Point >> CoordSyst_or_Point
```

Der Operator `>>` erfüllt diese Operation ebenfalls.

```
speed = soya.Vector(character, 0.0, 0.0, -1.0)
character.add_vector(speed)
```

Vektoren können vielfältig angewendet werden. Das obige Beispiel zeigt wie ein Objekt eine Einheit um eins weiter bewegt werden könnte. Dazu wird ein speed Vektor in dem `CoordSyst` des `character` konstruiert, welcher den z-Wert -1 hat. Die Bewegung erfolgt dann mit `add_vector`.

Wie Punkte sind auch Vektoren in der Szene nicht sichtbar und auch nicht als Kinder aufgelistet. Auch hier gibt es die gleichen Bewegungsoperationen wie bei CoordSyst.

Die `length` Methode gibt die Länge eines Vektors zurück.

```
vector = moon.vector_to(sun)
vector.set_length(1.0)
moon.add_vector(vector)
```

Mit Hilfe von `set_length` kann der Vektor auf eine bestimmte Länge skaliert werden, ohne seine Richtung zu verändern. Das obige Beispiel bewegt den Mond um eine Einheit Richtung Sonne.

```
print (earth >> sun).angle_to(earth >> moon)
```

Die Methode `angle_to` gibt den Winkel, wie gewohnt in Grad, zwischen zwei `CoordSyst` zurück. So gestaltet sich die Berechnung des Winkels zwischen der Sonne und den Mond sehr einfach. Wir behalten im Hinterkopf, dass `>>` ein Vektor zwischen zwei Punkten erschafft.

4.2.1 Das skalare Produkt

Das skalare oder innere Produkt[10][6][9] zweier Vektoren ergibt ein Skalar, also eine Gleitpunktzahl. Das lässt sich als Multiplikation der Länge der senkrechten Projektion des zweiten Vektor auf den ersten Vektor verstehen.

```
q1=soya.Vector(scene, 5, 0, 0)
q2=soya.Vector(scene, 0, 5, 0)
print q1.dot_product(q2)
```

Stehen zwei Vektoren im rechten Winkel aufeinander, so ergibt sich das Produkt zu 0.

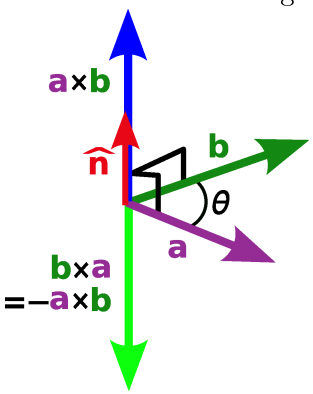
Wie wir sehen wird der englische Begriff `dot_product` für das skalare Produkt verwendet.

4.2.2 Das Kreuzprodukt

Das Kreuzprodukt[2][14] oder vektorielles bzw. äußeres Produkt ist ein Vektor der normal auf der von den Vektoren aufgespannten Ebene steht. Die Länge des Vektors entspricht der Fläche des Parallelogramms der zwei Vektoren.

```
a=soya.Vector(scene, 5, 0, 0)
b=soya.Vector(scene, 0, 5, 0)
axb=a.cross_product(b)
bxa=b.cross_product(a)
```

Hier wird auch der englische Begriff `cross_product` für das Kreuzprodukt verwendet.



Wie wir an der grafischen Illustration sehen, ist das Kreuzprodukt nicht kommutativ, der Ergebnisvektor kann auch in die Gegenrichtung zeigen. Die besondere Bedeutung der Rechenoperation ist die Bestimmung der Normalen auf Flächen die oftmals bei Rendering Methoden benötigt werden.

5 Verwaltung von Daten

5.1 Data-Dir

Soya liefert gleich eine Abstraktion zu Dateien mit um nicht bei jedem Modell, das geladen wird dem Programmierer vor die Aufgabe zu stellen auf dem System die richtigen Dateien finden zu müssen.

Stattdessen wird eine Liste von Data-Dirs, welcher vor der Initialisierung hinzugefügt werden übergeben.

```
import soya
soya.path.append ("/your/data/path")
soya.path.append ("/second/data/path")
soya.init("Data-Dir Example")
```

In jedem von diesem Data-Dir wird nach allem gesucht was in Soya benötigt wird. Dabei werden folgende Unterverzeichnisse verwendet.

images enthalten Bilder im PNG und JPEG Format für Texturen. Als Endungen sind .jpeg und .png zu verwenden.

materials enthalten Materialien für Soya (siehe 2.6). Die Dateiendungen sind .data. Sie werden aus den Bildern erzeugt.

models enthalten Modelle für Soya. Diese werden zum Laden von Modelle benötigt. Es funktioniert über das bekannte `soya.Model.get`. Die Dateiendungen sind .data.

```
sword_model = soya.Model.get("sword")
```

animated_models enthalten animierte Modelle für Soya. Dabei wird das Cal3D Format verwendet. Die Dateiendungen sind .caf, .cfg, .cmf, .crf und .csf.

world enthalten Welten für Soya. Hier können alle 3D Szenen oder auch einzelne Objekte (z.b. Body) geladen und abgespeichert werden.

blender enthalten Blenderdaten. Diese können nicht direkt von Soya3D geladen werden. Die verwendete Dateiendung dafür ist .blend.

sounds enthalten Musikdateien für Soya3D. Verwendet werden können OGG und WAV in den Dateiendungen .wav und .ogg.

fonts enthalten Schriftarten. Dabei sind TrueType Schriftarten mit der Endung .ttf erlaubt.

Das sind alle Objekte die Soya derzeit laden, speichern und automatisch konvertieren kann.

5.2 Dateiformate

Die Soya3D spezifischen Objekte, wie Materialien, Modelle und Welten werden durch *Serialisierung* gespeichert. Die Dateiformate werden deshalb durch den Serialisierer bestimmt. Diesen kann man folgendermaßen auswählen.

```
import cPickle, Cerealizer
set_file_format([cerealizer, cPickle], [cerealizer, cPickle])
```

cPickle ist dabei der Serialisierer welcher bei der Python Standarddistribution dabei ist, *Cerealizer* der bevorzugte für Soya3D, da er Sicherheitsprobleme vermeidet und komplett in Python geschrieben ist.

Durch den Aufruf `set_file_format` können die Serialisierer in einer bevorzugten Reihenfolge angegeben werden. Soll zuerst cPickle verwendet werden und dann cerealizer, so verwendet man:

```
import cPickle, Cerealizer
set_file_format(cPickle, cerealizer)
```

Will man nur einen Serialisierer angeben, so schreibt man diesen doppelt. Dann können die Dateien des anderen Serialisierer nicht mehr gelesen werden, was z.b. über Netzwerk erwünscht sein kann, so könnte man cPickle ausschließen:

```
import Cerealizer
set_file_format(cerealizer, cerealizer)
```

Cerealizer hat aber den Nachteil, dass jede Klasse bei ihm registriert werden muss. Während das bei den Standardklassen von Soya3D von selbst gemacht wird, müssen eigene Klassen erst registriert werden:

```
class YourWorld(soya.World):
cerealizer.register(YourWorld, SavedInAPathHandler(YourWorld))
```

`SavedInAPathHandler` ermöglicht dabei dass `YourWorld.get` funktioniert.

Beim Laden von Soya3D Modellen werden automatisch einige Konversationen durchgeführt. Dabei wird jedes Bild zu Material, die Blender Models zu World, World zu Model aber auch Blender Model zu AnimatedModel umgewandelt. Das bedeutet, dass wenn z.b. eine Textur geöffnet wird, wo es ein neueres Bild gleichen Namens gibt, so wird dieses Bild zu der Textur gewandelt. Dabei wird rekursiv gesucht, es wird also, wenn Blender installiert ist, auch von Blender zu Welt und dann zu Modell transformiert.

Es kann also jedes Modell von Blender für Soya verwendet werden und es wird sogar automatisch konvertiert. Wie man Models in Blender erzeugt, siehe Kapitel 10

Ist dieses Feature bei der Endversion des Produktes nicht mehr erwünscht, weil z.b. die Timestamps bei der Installation durcheinander gebracht werden, so kann man es deaktivieren:

```
soya.AUTO_EXPORTERS_ENABLED = 0
```

Ein üblicher Trick ist es das Feature von CVS, Subversion oder Git Repositories abhängig zu machen, wodurch für Entwickler automatisch konvertiert wird, nach der Distribution dies aber verhindert wird:

```
APPDIR = os.path.dirname(sys.argv[0])
# os.path.dirname(__file__) for a Python module
soya.AUTO_EXPORTERS_ENABLED = os.path.exists(os.path.join(APPDIR, ".git"))
```

5.3 Speichern und Laden von Daten

Szenen können somit auch im Spiel kreiert werden und dann gespeichert werden. Es können aber auch bereits gespeicherte Szenen von der Festplatte oder über das Netzwerk empfangen und dann angezeigt werden.

Wir wollen uns das Laden mit einem Beispiel genauer anschauen:

Listing 5.1: [basic-loadingfile-1.py]Laden von Dateien

```
import sys, os, os.path, soya
import cerealizer
execfile(os.path.join(os.path.dirname(sys.argv[0]),
    "basic-savingfile-cerealizer-1.py"))
```

Nach dem Laden der üblichen Module muss auch ceralizer importiert werden. In unseren Beispiel werden wir diesen Serialisierer verwenden. Als letzte Aktion wird das Programm ausgeführt welches die Datei speichert. Dieses Programm wird im Anschluss erklärt.

```
soya.init()
soya.path.append(os.path.join(os.path.dirname(sys.argv[0]), "data"))
scene = soya.World.load("a_scene_with_a_rotating_body")
```

Wir initialisieren Soya wie üblich, laden aber statt dem Modell Schwert die gewünschte Welt `a_scene_with_a_rotating_body` genannt. Sie ist auch in `./data/worlds/-a_scene_with_a_rotating_body.data` vorhanden, wenn basic-savingfile-cerealizer-1.py vorher erfolgreich ausgeführt werden konnte.

```
camera = soya.Camera(scene)
camera.z = 3.0
soya.set_root_widget(camera)
soya.MainLoop(scene).main_loop()
```

Soya3D hat hier leider eine Schwäche. Es ist nicht möglich Kameradaten abzuspeichern oder zu Laden. Wir müssen deshalb die Kamera manuell wie gewünscht platzieren.

Wir sehen, dass das Laden sich sehr einfach gestaltet und wollen uns jetzt dem Speichern einer Welt zuwenden.

Listing 5.2: [basic-savingfile-cerealizer-1.py]Speichern in Dateien

```
import sys, os, os.path, soya
import cerealizer
```

Auch für das Speichern dürfen wir nicht vergessen das Modul cerealizer zu laden.

```
class RotatingBody(soya.Body):
    def advance_time(self, proportion):
        soya.Body.advance_time(self, proportion)
        self.rotate_y(proportion * 5.0)
```

Wir kreieren den rotierenden Körper wie bereits gehabt. Man beachte, dass diese Rotationsinformation im späteren Objekt erhalten bleibt.

```
cerealizer.register(RotatingBody)
```

Cerealizer erfordert, dass Objekte registriert werden. Im obigen Beispiel funktioniert es durch das Ausführen des Programmes mit `execfile`. Hier müssen wir Hand anlegen und registrieren die Klasse mit dem Aufruf der Methode `register`.

```
if sys.argv[0].endswith("basic-savingfile-cerealizer-1.py"):
    soya.init()
    soya.path.append(os.path.join(os.path.dirname(sys.argv[0]), "data"))
    soya.set_file_format(cerealizer)
```

Die erste Zeile ermöglicht, dass der anschließende Block nur gestartet wird, wenn das Programm tatsächlich basic-savingfile-cerealizer-1.py heißt. Nach der Initialisierung wird `set_file_format` wie oben besprochen ausgeführt. Wir verwenden in diesem Fall cerealizer.

```
    scene = soya.World()
    sword_model = soya.Model.get("sword")
    sword = RotatingBody(scene, sword_model)
    light = soya.Light(scene)
    light.set_xyz(0.5, 0.0, 2.0)
    scene.filename = "a_scene_with_a_rotating_body"
    scene.save()
```

Dieser Teil speichert die Szene samt Schwert und Licht. Der Dateiname wird wie zuvor bereits verlangt mit `a_scene_with_a_rotating_body` festgelegt. `save` veranlasst das Serialisieren aller Objekte von der Szene ausgehend.

```
    camera = soya.Camera(scene)
    camera.z = 3.0
    soya.set_root_widget(camera)
    soya.MainLoop(scene).main_loop()
```

Da die Kamera leider sowieso nicht serialisiert werden kann, wird sie erst hier erschaffen und anschließend die Hauptschleife ausgeführt.

6 Interaktionen

Soya3D ist nicht nur für Visualisierungen hervorragend geeignet. Es ist auch sehr leicht, mit Soya3D Software auf Tastatur, Maus und Joystick zu interagieren.

Es werden direkt Keycodes von *SDL* verwendet. Das hat den Vorteil dass es viel Dokumentation darüber im Netz gibt, wirkt aber im Soya3D Code ein wenig befremdlich. Es wird darüber nachgedacht, auch eine Soya3D Variante für Interaktion bereitzustellen.

Die SDL Variante hat aber den Vorteil dass jede Hardware die durch die SDL Implementation unterstützt wird, sofort auch von Soya3D Programmen benutzt werden kann. Würde z.b. Wii Remote mit neuen Eventstrukturen eingefügt werden, könnte man es sofort verwenden, ohne dass Soya3D großartig modifiziert werden müsste.

6.1 Ereignistypen

Um auf die SDL Konstanten zugreifen zu können, ist folgender `import` notwendig:

```
import sys, os, os.path, soya, soya.sdlconst

event = soya.process_event()
```

`process_event` gibt ein Array mit unterschiedlicher Anzahl von Elementen zurück. Im ersten Feld steht der Ereignistyp (engl. eventtype). Hier ist eine Liste aller Ereignistypen.

Listing 6.1: [eventtypes.py]Eventtypen

```
(sdlconst.KEYDOWN, key, mods[, unicode_key])
(sdlconst.KEYUP, key, mods[, unicode_key])
(sdlconst.MOUSEMOTION, x, y, x_relative, y_relative, state)
(sdlconst.MOUSEBUTTONDOWN, button, x, y)
(sdlconst.MOUSEBUTTONUP, button, x, y)
(sdlconst.JOYAXISMOTION, axis, value)
(sdlconst.JOYBUTTONDOWN, button)
(sdlconst.JOYBUTTONUP, button)
(sdlconst.VIDEORESIZE, width, height)
(sdlconst.VIDEOEXPOSE)
(sdlconst.QUIT)
```

`unicode_key` ist allerdings nur vorhanden wenn, `soya.set_use_unicode(1)` aufgerufen wurde.

Mit Beistrichen abgetrennt, sieht man wieviele Felder der Typ jeweils hat. In den folgenden Kapiteln werden wir auf alle Typen genau eingehen.

Meistens will man Events in einer Schleife abarbeiten. Dazu bietet sich das Python `for` Konstrukt hervorragend an.

```
for event in soya.process_event():
    if event[0] == soya.sdlconst.<eventtype>:
        #process event from <eventtype>
```

Zuerst wird mit `if` festgestellt von welchem Typ der Event ist, um die richtige Anzahl der Elemente im Array zu wissen. Danach kommt eine Logik die auf diese Argumente reagiert.

Da hier auch Programmlogik stattfinden kann, sollte diese Schleife typischerweise in `begin_round` stehen.

6.2 Tastatur

Jede Taste sendet zwei Eventtypen KEYDOWN und KEYUP beim Drücken und Loslassen. Ohne Unicode können nur ASCII Zeichen sinnvoll empfangen werden und es gibt nur zwei Parameter, also ein Array mit 3 Werten.

Wir wollen die Schlange der ersten Beispiele mit der Tastatur steuern können.

Listing 6.2: [basic-5.py]Schlange mit Tastur steuern

```
def begin_round(self):
    soya.Body.begin_round(self)
```

Wie bei Ereignistypen bereits angeführt, platzieren wir das prozessieren der Events am günstigsten bei `begin_round` vom CaterpillarHead, da wir diesen steuern wollen, da die CaterpillarPiece ihm folgen.

```
for event in soya.process_event():
    if event[0] == soya.sdlconst.KEYDOWN:
```

Auch diesen Teil kennen wir bereits, in der Schleife werden alle noch nicht verarbeiteten Events abgearbeitet. Wir stellen zuerst einmal den Typ des Events fest um Laufzeitfehler zu vermeiden und KEYDOWN getrennt von KEYUP behandeln zu können.

Folgend wollen wir mit den Cursortasten die Geschwindigkeit regulieren können. Wird die Cursortaste nach oben gedrückt, so wird eine negative Geschwindigkeit angenommen, sonst eine positive.

```
if   event[1] == soya.sdlconst.K_UP:
    self.speed.z = -0.2
elif event[1] == soya.sdlconst.K_DOWN:
    self.speed.z =  0.1
```

Das Lenken hingegen wird durch den Rotationswinkel vollzogen. Pro Event wird sich der Kopf um 10° drehen.

```
elif event[1] == soya.sdlconst.K_LEFT:
    self.rotation_y_speed =  10.0
elif event[1] == soya.sdlconst.K_RIGHT:
    self.rotation_y_speed = -10.0
```

Bis jetzt war es nicht möglich, das Programm regulär zu beenden. Hier wollen wir die Möglichkeit mit der Taste "q" und "Esc" bereitstellen. Dazu beenden wir einfach die Hauptschleife wenn eine solche Taste gedrückt wird:

```
        elif event[1] == soya.sdlconst.K_q:
            soya.MAIN_LOOP.stop()
        elif event[1] == soya.sdlconst.K_ESCAPE:
            soya.MAIN_LOOP.stop()
```

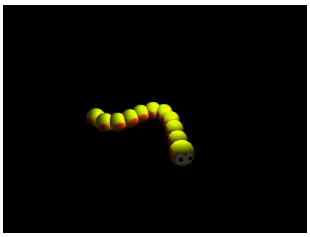
Mit KEYUP wollen wir die Werte wieder auf 0.0 zurücksetzen, damit sich die Schlange nicht weiter bewegt:

```
    elif event[0] == soya.sdlconst.KEYUP:
        if   event[1] == soya.sdlconst.K_UP:
            self.speed.z = 0.0
        elif event[1] == soya.sdlconst.K_DOWN:
            self.speed.z = 0.0
        elif event[1] == soya.sdlconst.K_LEFT:
            self.rotation_y_speed = 0.0
        elif event[1] == soya.sdlconst.K_RIGHT:
            self.rotation_y_speed = 0.0
```

Hier wollen wir kurz einen anderen Eventtyp vorstellen.

QUIT wird ausgelöst, wenn versucht wurde die SDL Applikation zu schließen. Damit kann nun endlich der Knopf für das Schließen des Windowmanagers verwendet werden:

```
    elif event[0] == soya.sdlconst.QUIT:
        soya.MAIN_LOOP.stop()
self.rotate_y(self.rotation_y_speed)
```



Wir wollen kurz resümieren. Es gibt Konstanten die für `soya.process_event()` verwendet werden. KEYDOWN gibt an, dass eine beliebige Taste hinuntergedrückt wurde, KEYUP gibt ein dazugehöriges Loslassen an.

`K_UP`, `K_DOWN`, `K_LEFT` und `K_RIGHT` sind die Tastendrücke von den Cursortasten. `K_ESCAPE` wird ausgelöst wenn die Esc Taste gedrückt wird. Zudem sind auch noch alle Groß- und Kleinbuchstaben vorhanden, z.b. q mit `K_q`, aber auch Zahlen, z.b. 1 mit `K_q` und Funktionstasten, z.b. F1 mit `K_F1`.

Andere Keycodes mit selbsterklärender Bedeutung sind `K_FIRST`, `K_BACKSPACE`, `K_TAB`, `K_CLEAR`, `K_RETURN`, `K_PAUSE`, `K_ESCAPE`, `K_SPACE`, `K_EXCLAIM`, `K_QUOTEDBL`, `K_HASH`, `K_DOLLAR`, `K_AMPERSAND`, `K_QUOTE`, `K_LEFTPAREN`, `K_RIGHTPAREN`, `K_ASTERISK`, `K_PLUS`, `K_COMMA`, `K_MINUS`, `K_PERIOD`, `K_SLASH`, `K_COLON`, `K_SEMICOLON`, `K_LESS`, `K_EQUALS`, `K_GREATER`, `K_QUESTION`, `K_AT`, `K_LEFTBRACKET`, `K_BACKSLASH`, `K_RIGHTBRACKET`, `K_CARET`, `K_UNDERSCORE`, `K_BACKQUOTE`, `K_DELETE`, `K_KP_PERIOD`, `K_KP_DIVIDE`, `K_KP_MULTIPLY`, `K_KP_MINUS`, `K_KP_PLUS`, `K_KP_ENTER`, `K_KP_EQUALS`, `K_UP`, `K_DOWN`, `K_RIGHT`, `K_LEFT`, `K_INSERT`, `K_HOME`, `K_END`, `K_PAGEUP` und `K_PAGEDOWN`.

Zusätzlich gibt es noch das Bitfeld für die Modifier wie Shift, Alt, Num, Caps,... Bei einem Bitfeld ist oftmals nicht sinnvoll mit `==` zu arbeiten, da verschiedene andere Mo-

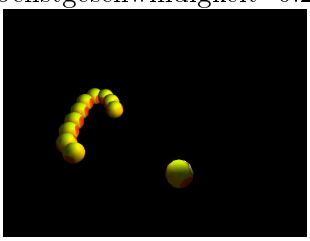
difier möglicherweise gesetzt sind, diese aber nicht relevant sind. Stattdessen verwendet man `&` welches nur genau dieses eine Bit überprüft.

Wir wollen unserer Schlange ein `speed up` verpassen wenn die linke Shifttaste vor und zusammen mit dem Cursor Up gedrückt wird. Anderenfalls belassen wir es auf die vorige Geschwindigkeit.

Listing 6.3: [basic-keyboard-mod.py]Tastatursteuerung mit Modifier

```
        if   event[1] == soya.sdlconst.K_UP:
            if event[2] & soya.sdlconst.MOD_LSHIFT:
                print "speed up"
                self.speed.z = -0.8
            else:
                self.speed.z = -0.2
```

Wie wir sehen fliegt nun der Kopf weg, weil die anderen Körperteile nur mit der Höchstgeschwindigkeit -0.2 folgen.



Die Modifier sind `MOD_NONE`, `MOD_LSHIFT`, `MOD_RSHIFT`, `MOD_SHIFT`, `MOD_LCTRL`, `MOD_RCTRL`, `MOD_CTRL`, `MOD_LALT`, `MOD_RALT`, `MOD_ALT`, `MOD_LMETA`, `MOD_RMETA`, `MOD_META`, `MOD_NUM`, `MOD_CAPS`, `MOD_MODE` und `MOD_RESERVED`.

Die Tasten sind aber auch als Keycodes ansprechbar. Damit kann man sie aber nur einmal einlesen wenn sie gedrückt bzw. losgelassen werden. Die Modifier hingegen bleiben fortlaufend bestehend, eine Eigenschaft die bei mehreren event Routinen sehr wichtig ist. Die Keycodes sind `K_NUMLOCK`, `K_CAPSLOCK`, `K_SCROLLOCK`, `K_RSHIFT`, `K_LSHIFT`, `K_RCTRL`, `K_LCTRL`, `K_RALT`, `K_LALT`, `K_RMETA`, `K_LMETA`, `K_LSUPER`, `K_RSUPER`, `K_MODE`, `K_COMPOSE`, `K_HELP`, `K_PRINT`, `K_SYSREQ`, `K_BREAK`, `K_MENU`, `K_POWER`, `K_EURO` und `K_UNDO`.

6.3 Maus

In dem letzten Basic Tutorial wird noch betrachtet wie wir auf die Maus zugreifen können. Auch diese Aktion ist einfach gehalten:

Listing 6.4: [basic-6.py]Steuerung mit der Maus

```
def begin_round(self):
    soya.Body.begin_round(self)
    for event in soya.process_event():
        if event[0] == soya.sdlconst.MOUSEMOTION:
            self.mouse_x = event[1]
            self.mouse_y = event[2]
    mouse_pos = camera.coord2d_to_3d(self.mouse_x,
        self.mouse_y, (self % camera).z)
    mouse_pos.convert_to(scene)
```

```
mouse_pos.y = 0.0
self.speed.z = -self.distance_to(mouse_pos)
self.look_at(mouse_pos)
```

Dabei wird das event MOUSEMOTION ausgelöst. Die x und y Koordinaten liegen in dem Array event, welches von `soya.process_event()` zurückgegeben wird. Allerdings müssen die Koordinaten noch in die dritte Dimension gebracht werden, da sie als Pixelwerte vorliegen. Dazu wird `coord2d_to3d` verwendet. Der z-Wert ist allerdings unbekannt. Sollte er fehlen, wird er oftmals mit -1 angenommen.

Es ist auch möglich von 3D zu 2D Koordinaten umzurechnen. Dafür kann die Methode `coord3d_to_2d` verwendet werden.

6.4 Joystick

Auch der Joystick kann ähnlich durch die SDL Konstanten angesprochen werden. Folgende Eventtypen existieren für den Joystick `JOYAXISMOTION`, `JOYBALLMOTION`, `JOYHATMOTION`, `JOYBUTTONDOWN` und `JOYBUTTONUP`.

Das folgende Beispiel existiert nicht in den Tutorials aber schon im Anhang, er ist von Balazar Brothers (siehe 15.5 kopiert. Dieser Code setzt Joystick Events zu Events die die Cursortasten produzieren würden um. Der Code ist nicht für sich alleine ausführbar.

Dabei wird `event_type`, `event_key` und `event_mod` verwendet um den Ereignistyp, den Key und den Modifier zu setzen. Dieser Ansatz hat den Vorteil dass es nur einen Code geben muss der sowohl Joystick als auch Tastatur prozessiert.

Listing 6.5: [joystick.py]Joystick zu Tastatur Event Konverter

```
for event in events:
  if    event[0] == sdlconst.JOYAXISMOTION:
```

Die Hauptschleife für events wird wie gehabt mit `for` realisiert. Jetzt reagieren wir aber auf Bewegungen der Hauptachse des Joysticks.

```
if    event[1] == 0:
  if    event[2] < 0:
 event_type = sdlconst.KEYDOWN
 event_key = sdlconst.K_LEFT
  elif event[2] > 0:
 event_type = sdlconst.KEYDOWN
 event_key = sdlconst.K_RIGHT
  else:
 event_type = sdlconst.KEYUP
 event_key = sdlconst.K_LEFT # XXX or right
```

Wie wir an der Übersichtstabelle (siehe Kapitel 6.1) erkennen können, steht der erste Parameter dafür welche Achse bewegt wird und der zweite der Wert gibt die Richtung an. Als Erstes überprüfen wir ob es die y-Achse ist, indem geschaut wird ob der erste Parameter gleich 0 ist.

Wenn sie es ist, wird überprüft ob der Wert kleiner oder größer 0 ist. Dieser Wert gibt die Auslenkung vom Zentrum an. Während im Zentrum der Wert 0 sein sollte (Joystickkalibration), sind die Maximalauslenkungen von -32767 bis 32767.

Somit werden wir für negative Werte den `event_key` auf `K_LEFT` setzen, für positive `K_RIGHT`. Hier entsteht ein Nachteil der Methode auf Tastaturdrücke umzurechnen, es geht die Information verloren, wie stark der Joystick ausgelenkt wurde.

```
elif event[1] == 1:
  if   event[2] < 0:
 event_type = sdlconst.KEYDOWN
 event_key = sdlconst.K_UP
  elif event[2] > 0:
 event_type = sdlconst.KEYDOWN
 event_key = sdlconst.K_DOWN
  else:
 event_type = sdlconst.KEYUP
 event_key = sdlconst.K_UP # XXX or down
```

Bei der anderen Achse wird gleich vorgegangen. Die x-Achse hat den Wert 1, mit ihr kann man dann die Cursortasten vor und zurück auslösen.

```
elif event[0] == sdlconst.JOYBUTTONDOWN:
  event_type = sdlconst.KEYDOWN
  event_key  = str(event[1])
  event_mod  = 0
```

Nun müssen wir noch auf den Joystickknopf reagieren. Als `event_key` übergeben wir den Buttonwert von dem ausgelösten Event. Dabei setzen wir aber die Modifier auf 0.

```
elif event[0] == sdlconst.JOYBUTTONUP:
  event_type = sdlconst.KEYUP
  event_key  = str(event[1])
  event_mod  = 0
```

Wird die Joysticktaste losgelassen verfahren wir gleich.

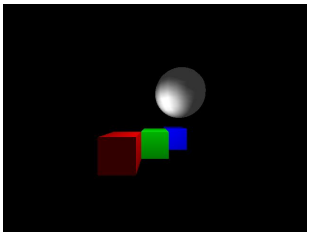
```
else:
  event_type = event[0]
  if len(event) > 1:
    event_key  = event[1]
    if len(event) > 3: event_mod = event[2]
    else:              event_mod = 0
```

Als letzten Fall werden die Events einfach weiter übergeben.

6.5 Drag and Drop

Für die Maus gibt es neben der Steuerung von Figuren auch noch das wichtige Einsatzgebiet Drag und Drop, welches mit Soya3D sehr einfach zu realisieren ist. Raypicking (siehe Kapitel 11) benötigen wir hier um zu erkennen auf welchen Gegenstand wir klicken.

Wir betrachten ein Beispiel in der wir einige Würfel und eine Kugel mit der Maus bewegen können.



Listing 6.6: [dragdrop.py]Drag and Drop

```
import sys, os, os.path, soya, soya.cube, soya.sphere, soya.sdlconst
soya.init()
soya.path.append(os.path.join(os.path.dirname(sys.argv[0]), "data"))
scene = soya.World()
class DragDropWorld(soya.World):
    def __init__(self, parent):
        soya.World.__init__(self, parent)
        self.dragdroping = None
        self.impact = None
```

Wir initialisieren `dragdroping` und `impact` mit None, welche wir später brauchen werden. Wie üblich werden die Events in `begin_round` prozessiert.

```
    def begin_round(self):
        soya.World.begin_round(self)
        for event in soya.process_event():
            if    event[0] == soya.sdlconst.MOUSEBUTTONDOWN:
                mouse = camera.coord2d_to_3d(event[2], event[3])
                result = self.raypick(camera, camera.vector_to(mouse))
```

Wir müssen hier auf das Event MOUSEBUTTONDOWN reagieren. Die zuvor gewonnen 3D Koordinaten werden dem `raypick` Aufruf übergeben. Wir schneiden somit von der Kamera mit einem Vektor zu den Mauskoordinaten.

```
                if result:
                    self.impact, normal = result
                    self.dragdroping = self.impact.parent
                    self.impact.convert_to(camera)
                    self.old_mouse = camera.coord2d_to_3d(event[2],
                        event[3], self.impact.z)
```

Wenn es ein Ergebnis gibt, setzen wir `self.dragdroping` auf `result.parent` das ist der Körper mit dem geschnitten wurde. Mehr Details dazu im Kapitel 11. Wir speichern uns aber auch die Mauskoordinaten in `self.old_mouse`.

```
            elif event[0] == soya.sdlconst.MOUSEBUTTONUP:
                self.dragdroping = None
```

Wird die Maustaste wieder losgelassen so stoppen wir das Drag and Drop.

```
            elif event[0] == soya.sdlconst.MOUSEMOTION:
                if self.dragdroping:
                    new_mouse = camera.coord2d_to_3d(event[1],
                              event[2], self.impact.z)
                    self.dragdroping.add_vector(
                              self.old_mouse.vector_to(new_mouse))
                    self.old_mouse = new_mouse
```

Der wichtigste Teil findet bei Bewegung der Maus statt. Hier wird ein Vektor von der alten zu der neuen Mauskoordinate berechnet. Dieser Vektor wird zu dem Objekt welches über `self.dragdroping` referenziert ist hinzuaddiert.

Damit ist der Hauptteil erledigt. Wir kreieren eine Welt, die eine Instanz unserer DragDropWorld Klasse ist und erschaffen ein paar Farben die wir dann gleich benötigen:

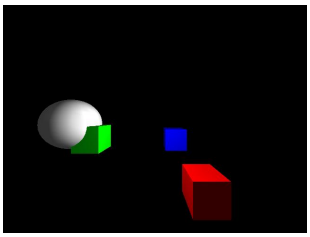
```
world = DragDropWorld(scene)
red   = soya.Material(); red   .diffuse = (1.0, 0.0, 0.0, 1.0)
green = soya.Material(); green.diffuse = (0.0, 1.0, 0.0, 1.0)
blue  = soya.Material(); blue  .diffuse = (0.0, 0.0, 1.0, 1.0)
```

Diese Welt kann wie üblich verwendet werden, die Drag and Drop Implementation ist transparent für die Objekte die in die Welt gesetzt werden und müssen auch nichts weiter außer `add_vector` unterstützen.

Nun fügen wir alle gewünschten Objekte in die Welt hinzu:

```
soya.Body(world, soya.cube.Cube(None, red   ).to_model()).set_xyz(-1.0, -1.0, 1.0)
soya.Body(world, soya.cube.Cube(None, green ).to_model()).set_xyz( 0.0, -1.0, 0.0)
soya.Body(world, soya.cube.Cube(None, blue  ).to_model()).set_xyz( 1.0, -1.0, -1.0)
soya.Body(world, soya.sphere.Sphere().to_model()).set_xyz(1.0, 1.0, 0.0)
```

und können nun die Würfel und die Kugel bewegen:



7 Sound

Musik und Geräuschkulissen haben besondere Bedürfnisse in 3D Anwendungen. Es soll der Eindruck gewahrt bleiben dass der Ton von der Stelle kommt wo er auftritt und unter Umständen je nach dazwischen liegenden Materialien gedämpft klingen. In diesem Kapitel werden wir bereits Gelerntes des Kapitel Ereignisse 6 benötigen.

7.1 Initialisierung

Soya startet defaultmäßig ohne eine Möglichkeit Ton von sich zu geben. Um grundsätzlich Sound zu initialisieren muss Soya folgendermaßen initialisiert werden.

```
soya.init ("App_with_sound", sound = 1)
```

Die Kamera hat dabei in der Szene neben der Funktionalität als Auge auch die des Ohres. Will man dieses Verhalten nicht, oder bei mehreren Kameras eine bestimmte Auswählen, so verwendet man:

```
camera.listen_sound = 0
```

Dieses Ohr kann beliebig viele `SoundPlayer` wahrnehmen. Dabei handelt es sich um Untertypen von `CoordSyst`.

```
sound = soya.Sound.get("my_sound.wav")
sound_player = soya.SoundPlayer(parent, sound)
music_player = soya.SoundPlayer(parent, sound, loop = 1, play_in_3D = 0)
```

Der Konstruktor von `Sound` kann .wav und .ogg Dateien, vom Data-Dir siehe Kapitel 5.1, laden. Dabei wird nicht die ganze Datei auf einmal geladen, sondern als Stream immer notwendige Teile angefordert. Um .ogg Dateien spielen zu können sind die Module PyOgg und PyVorbis notwendig. Dieses Format wird vor allem für längere Musikstücke sehr empfohlen, da es schneller arbeitet und weniger Speicherplatz verbraucht.

Der SoundPlayer in der letzten Zeile spielt das Geräusch immer wieder ab. Zudem wurde dreidimensionaler Sound deaktiviert, somit kann es als Hintergrundmusik verwendet werden.

Beim Speichern einer Welt wird automatisch darauf Rücksicht genommen, an welcher Stelle die SoundPlayer gerade waren. Nach dem Laden werden sie in etwa von dieser Stelle wieder weiterspielen.

```
print soya.get_sound_volume()
soya.set_sound_volume(0.5)
```

Die Tonstärke kann jederzeit abgefragt und gesetzt werden. Der dafür verwendete Wert reicht von 0.0 (kein Ton) bis 1.0 (default und maximal). Für die `SoundPlayer` kann der Ton mit `gain` gesetzt werden.

7.2 Verwendung

Listing 7.1: [sound-1.py]Im Kreis drehender Würfel

```
import soya, math, soya.cube as cube
soya.init(sound = 1)
soya.set_sound_volume(1.0)
```

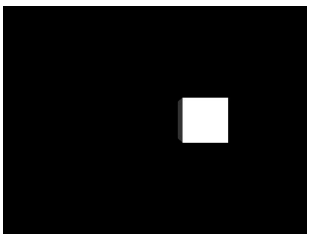
Bei der Initialisierung muss wie zuerst erklärt der Ton eingeschalten werden. Die Lautstärke ist zwar per default 1.0, hier wäre aber die richtige Stelle um es für einen Benutzer konfigurierbar zu machen.

```
class NoisyCube(soya.World):
    def __init__(self, parent, model):
        soya.World.__init__(self, parent, model)
        self.angle = 3.0
    def advance_time(self, proportion):
        soya.check_al_error()
        soya.World.advance_time(self, proportion)
        soya.check_al_error()
        self.angle += 0.04 * proportion
        self.set_xyz(5.0 * math.cos(self.angle), 0.0,
                5.0 * math.sin(self.angle))
```

Die Klasse lärmender Würfel ist der wichtigste Teil. Dieses Objekt bewegt sich im Kreis um die Kamera. Da es sich um eine Welt handelt, können Kinder hinzugefügt werden, z.b. `SoundPlayer`:

```
cube = NoisyCube(scene, cube.Cube().to_model())
sound = soya.Sound.get("test.wav")
sound_player = soya.SoundPlayer(cube, sound, loop = 1)
```

In der ersten Zeile wird bereits eine Welt und ein Modell `cube` erstellt. Danach wird der Ton geladen und der `SoundPlayer` initialisiert. Zu Beachten ist dabei dass `cube` als Elternteil verwendet wird. Dadurch wird sichergestellt, dass sich der `SoundPlayer` immer dort befindet wo der lärmende Würfel sich gerade aufhält.



7.3 Stereo

Nun wollen wir einen lärmenden `Cube` den wir mit der Maus nach links und rechts steuern können. Dazu müssen wir eine Welt erschaffen, deren Kinder ein `Model` und ein `SoundPlayer` sein werden.

Die Initialisierung hat zusätzlich noch einen Speedvektor:

Listing 7.2: [sound-2.py]Mit Maus gesteuerter Würfel

```
class NoisyCube(soya.World):
    def __init__(self, parent, model):
        soya.World.__init__(self, parent, model)
        self.speed              = soya.Vector(self, 0.0, 0.0, 0.0)
        self.rotation_y_speed   = 0.0
        self.mouse_x            = 0
        self.mouse_y            = 0
```

Hier wird die Welt in die Richtung wo die Maus ist hinbewegt. Dazu muss wie im Kapitel 6 besprochen die 2D Werte der Maus zu einem 3D Wert umgerechnet werden.

```
    def begin_round(self):
        soya.World.begin_round(self)
        for event in soya.process_event():
            if event[0] == soya.sdlconst.MOUSEMOTION:
                self.mouse_x = event[1]
                self.mouse_y = event[2]
        mouse_pos = camera.coord2d_to_3d(self.mouse_x,
                self.mouse_y, (self % camera).z)
        mouse_pos.convert_to(scene)
        mouse_pos.y = 0.0
        self.speed.z = -self.distance_to(mouse_pos)
        self.look_at(mouse_pos)
```

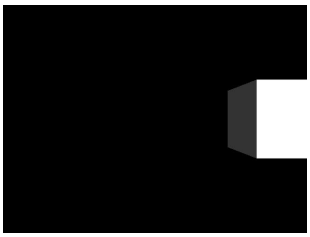
Dieser Teil bewegt den Cube wenn die Zeit verrinnt.

```
    def advance_time(self, proportion):
        soya.World.advance_time(self, proportion)
        self.add_mul_vector(proportion, self.speed)
    def ended(self):
        print "ended"
```

Zum Schluss müssen wir wie besprochen ein Modell `Cube` und einen `SoundPlayer` in die Welt einfügen, der dann mit der Maus bewegt werden kann.

```
cube = NoisyCube(scene, cube.Cube().to_model())
cube.set_xyz(0,0,-3.0)
sound = soya.Sound.get("test.wav")
sound_player = soya.SoundPlayer(cube, sound, loop = 1)
```

Nun können wir den Cube nach links und rechts verschieben und dabei beobachten wie die Lautstärke an den linken und rechten Boxen höher und niedriger wird.



7.4 Dopplereffekt

Einen weiteren Effekt können wir beobachten, wenn wir den Cube mit der Tastatur steuern.

Die Initialisierung des lärmenden Würfel gestaltet sich ähnlich wie bei dem vorigen Beispiel:

Listing 7.3: [sound-3.py]Mit Tastatur gesteuerter Würfel

```
class NoisyCube(soya.World):
    def __init__(self, parent, model):
        soya.World.__init__(self, parent, model)
        self.speed = soya.Vector(self, 0.0, 0.0, 0.0)
        self.speed.z=-0.2
        self.rotation_y_speed = 0.0
```

Hier muss allerdings auf die Tastaturevents reagiert werden. Hier ist eine einfache Steuerung mit den Cursortasten implementiert:

```
    def begin_round(self):
        soya.World.begin_round(self)
        for event in soya.process_event():
            #print event[0]
            if event[0] == soya.sdlconst.KEYDOWN:
                if event[1] == soya.sdlconst.K_LEFT:
                    self.rotation_y_speed =  10.0
                elif event[1] == soya.sdlconst.K_RIGHT:
                    self.rotation_y_speed = -10.0
            elif event[0] == soya.sdlconst.KEYUP:
                if event[1] == soya.sdlconst.K_LEFT:
                    self.rotation_y_speed = 0.0
                elif event[1] == soya.sdlconst.K_RIGHT:
                    self.rotation_y_speed = 0.0
                elif event[1] == soya.sdlconst.K_q:
                    soya.MAIN_LOOP.stop()
                elif event[1] == soya.sdlconst.K_ESCAPE:
                    soya.MAIN_LOOP.stop()
            elif event[0] == soya.sdlconst.QUIT:
                soya.MAIN_LOOP.stop()
        self.rotate_y(self.rotation_y_speed)
```

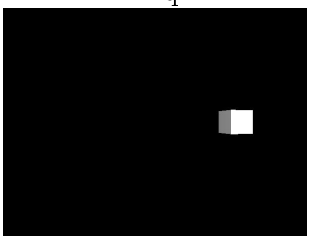
Im `advance_time` muss dann nur der Vektor für die Geschwindigkeit hinzuaddiert werden. Wichtig ist, dass beachtet wird, dass wir hier von einer Welt und nicht einen Body ableiten, wir müssen deshalb `soya.World.advance_time` aufrufen:

```
    def advance_time(self, proportion):
        soya.World.advance_time(self, proportion)
        self.add_mul_vector(proportion, self.speed)
```

Der letzte Teil hingegen wird wieder gleich implementiert:

```
cube = NoisyCube(scene, cube.Cube().to_model())
cube.set_xyz(0,0,-3.0)
sound = soya.Sound.get("test.wav")
sound_player = soya.SoundPlayer(cube, sound, loop = 1)
```

Hier können wir den **Dopplereffekt** wahrnehmen. Bewegt sich der Cube auf uns zu, so wird die Frequenz höher, bewegt er sich von uns weg, so wird die Frequenz tiefer.



8 ODE

8.1 Einführung

Open Dynamics Engine oder kurz ODE ist eine Open Source Physik-Engine. Diese kann dazu verwendet werden um Kollisionen zu erkennen oder Körperdynamik zu simulieren. Dabei handelt es sich jedoch nur um eine Engine, die selbst jedoch keine graphische Ausgabe hat. Es wäre zwar möglich die Welt mit ODE zu simulieren und die Berechnungen mit Soya3D graphisch darzustellen, aber Soya3D hat mit der Version 0.13 ODE integriert und somit kann man sich diesen Umweg ersparen. Wie ODE unter Soya3D verwendet werden kann lässt sich am besten mit Beispielen erklären.

8.2 Kollisionen in ODE

8.2.1 Frontalzusammenstoß

Bei diesen Beispielen lassen wir zwei Kugeln gegeneinander fliegen, so dass sie nach einer gewissen Zeit kollidieren und sich exakt in der Mitte treffen. Dadurch ändern sie ihre Richtung um 180 Grad. Die Kugeln modellieren wir als Raupenkopf um ihre Richtung erkennen können.

Listing 8.1: [ode-collision-1-base.py]Einfache Kollision

```
import sys, os
import soya

soya.init("collision-1-base",width=1024,height=768)
soya.path.append(os.path.join(os.path.dirname(sys.argv[0]), "data"))

scene = soya.World()
head_model = soya.Model.get("caterpillar_head")

heads = ( soya.Body(scene,head_model), soya.Body(scene,head_model))
```

Bis jetzt ist alles bekannt (siehe 2.4). Wir benutzen als Modell den Kopf einer Raupe (`caterpillar_head`). Davon erzeugen wir zwei Stück.

```
for head in heads:
    head.mass = soya.SphericalMass()
```

ODE muss wissen, wie die Masse verteilt ist. Hier geben wir an, dass es sich um zwei kugelförmige Massen handelt.

```
soya.GeomSphere(heads[0],1.2)
soya.GeomSphere(heads[1],1.2)
```

Hier werden die geometrischen Eigenschaften der Kugel bekannt gegeben. Diese sind notwendig damit die Kollision an der richtigen Position, also am äußeren Rand der Kugel und nicht in ihrem Zentrum stattfindet.

```
heads[0].x = -25
heads[1].x = 25
heads[0].look_at(heads[1])
heads[1].look_at(heads[0])
```

Nun platzieren wir die Raupen und richten sie so aus, dass sie einander betrachten.

```
for head in heads:
    head.add_force(soya.Vector(head,0,0,-500))
```

Damit sich die Raupen aufeinander zu bewegen müssen wir eine Kraft auf sie ausüben. Wir verwenden für beide die gleiche Kraft, sodass sie sich gleich schnell bewegen.

```
light = soya.Light(scene)
light.set_xyz(0, 15,0)
camera = soya.Camera(scene)
camera.set_xyz(0,0,50)
soya.set_root_widget(camera)
ml = soya.MainLoop(scene)
ml.main_loop()
```

Zum Schluss setzen wir hier nur Licht und Kamera und führen die Hauptschleife aus.

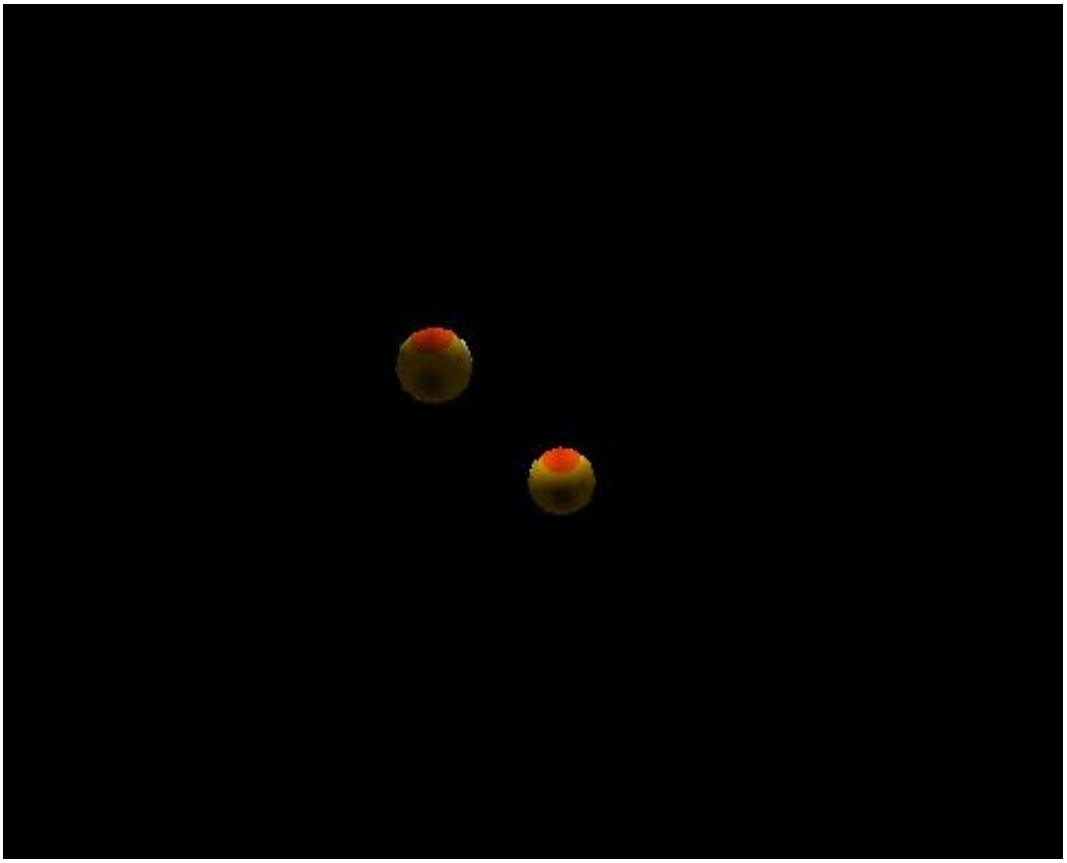
8.2.2 Versetzter Frontalzusammenstoß

Bis jetzt haben die beiden Raupen bei ihrem Zusammenstoß nur ihre Richtung gewechselt und sind langsamer geworden. Dies wäre noch einfach zum selber modellieren gewesen. Wir wollen uns allerdings ansehen, wie es aussieht wenn die beiden Kugeln leicht versetzt miteinander kollidieren. Wer gelegentlich Billard spielt weiß, dass die Kugeln dann schräg abgelenkt werden. Außerdem wirkt zusätzlich ein Drehmoment, so dass die Kugeln beginnen sich zu drehen. Durch geringfügige Änderungen am vorigen Beispiel können wir den Zusammenstoß visualisieren. Durch die Verwendung von Raupenköpfen sehen wir dabei auch die Drehung der Kugeln.

Listing 8.2: [ode-collision-2-base.py]Versetzter Zusammenstoß

```
heads[0].set_xyz(-25,-0.5,-0.7)
heads[1].set_xyz(25,0.4,0.8)
```

Anstatt wie beim vorigen Beispiel nur den x-Wert von head zu setzen, setzen wir auch y und z. Dies geschieht am einfachsten mit der set_xyz-Methode.



8.2.3 Einfluss der Masse

Ein schwererer Körper wird bei einer Kollision weniger abgelenkt, als der leichtere. Dafür wird mehr Kraft zur Beschleunigung benötigt. Im Gegensatz zum richtigen Leben ist es bei Soya3D sehr einfach mehr Kraft auf einen Körper wirken zu lassen. Wir können durch geringfügige Änderungen am vorigen Beispiel den Zusammenstoß unterschiedlich schwerer Körper visualisieren.

Listing 8.3: [ode-collision-3-mass-influence.py]Zusammenstoß unterschiedlicher Massen

```
heads[0].mass = soya.SphericalMass()
heads[1].mass = soya.SphericalMass(8)
```

Anstatt wie in den vorigen beiden Beispielen jedem head die gleiche Masse zu geben, geben wir hier die Massen einzeln an. Der zweite Kopf hat die 8-fache Masse des ersten, da SphericalMass() standardmäßig auf 1 initialisiert wird.

```
heads[0].add_force(soya.Vector(heads[0],0,0,-1000))
heads[1].add_force(soya.Vector(heads[1],0,0,-8000))
```

Das gleiche gilt für die Kräfte, welche auf die beiden Kugeln wirken. Damit die beiden unterschiedlich schweren Kugeln sich gleich schnell bewegen, müssen wir sie einer Kraft proportional zu ihrer Masse aussetzen.

Soll ein Körper sich nicht durch eine Kollision aus der Bahn bringen lassen, so kann man dies durch setzen der pushable-Eigenschaft erreichen. Im folgenden ändern wir unser Beispiel so um, dass der leichtere Körper die pushable-Eigenschaft erhält und damit seine Bahn beibehält. Dazu fügen wir folgende Zeile hinzu:

Listing 8.4: [ode-collision-4-pushable.py]Stoßbare Objekte

```
heads[0].pushable = False
```

Tritt eine Kollision ein, so möchte man häufig darauf reagieren. Soya3D bietet die Möglichkeit dafür mit sogenannten Hit Functions. Die Klasse soya.Body ruft bei einer Kollision die hit-Methode auf. Diese ist aber üblicherweise leer. Wir können sie allerdings beim Vererben überschreiben.

Listing 8.5: [ode-collision-5-hit-func.py]Hit Functions

```
class SpeakingHead(soya.Body):
    head_model = soya.Model.get("caterpillar_head")
    def __init__(self,parent,name):
        soya.Body.__init__(self,parent,self.head_model)
        self.name = name
    def hit(self,*args,**kargs):
        print "<%s> outch I'm it !"%self.name
```

Und natürlich müssen wir nun auch unsere SpeakingHeads verwenden anstelle der soya.Bodys.

```
heads = (
    SpeakingHead(scene,"lili"),
    SpeakingHead(scene,"pipo"))
```

Nun wird bei jeder Kollision eine Nachricht auf die Standardausgabe ausgegeben.

8.2.4 Mehrere Objekte

Bis jetzt haben wir nur zwei Objekte kollidieren lassen. Es ist aber problemlos möglich dies mit mehreren zu tun. Im folgenden wollen wir 4 Objekte erzeugen, die miteinander kollidieren und sich bei ihrem Gegenüber dafür entschuldigen. Außerdem sollen die einzelnen Raupen unterschiedliche Varianten von Entschuldigungen wählen. Um den Namen, des Objekts zu erhalten, mit dem ein Objekt kollidiert, können wir in der hit-Methode den ersten optionalen Parameter other verwenden. Dieser ist eine Referenz auf das Objekt mit dem wir kollidiert sind.

Listing 8.6: [ode-collision-6-hit-func-2-other.py]Mehrere Kollisionen

```
class PoliteHead(soya.Body):
    head_model = soya.Model.get("caterpillar_head")
    sentences = [
        "<%s> Oups sorry *%s*!",
        "<%s> Damned I hit *%s*",
        "<%s> excuse me *%s*. I hope I didn't hurt you",
        "<%s> *%s*! please mind you step !",
        "<%s> ho I did see you *%s* how. are you ?",
        ]
    def __init__(self,parent,name):
```

```
        soya.Body.__init__(self,parent,self.head_model)
        self.name = name
    def hit(self,other,*args,**kargs):
        print choice(self.sentences)%(self.name,other.name)
```

Weiters erzeugen wir nun 4 Objekte und weisen ihnen eine Masse zu:

```
head_model = soya.Model.get("caterpillar_head")
head_a = PoliteHead(scene,"Pat")
head_a.mass = soya.SphericalMass(a_mass,1,"total_mass")
head_b = PoliteHead(scene,"Lili")
head_b.mass = soya.SphericalMass(3,1,"total_mass")
head_c = PoliteHead(scene,"Sam")
head_c.mass = soya.SphericalMass(c_m,1,"total_mass")
head_d = PoliteHead(scene,"Mike")
head_d.mass = soya.SphericalMass(d_m,1,"total_mass")
```

Außerdem müssen wir die Objekte platzieren und Kräfte darauf wirken lassen:

```
head_a.add_force(soya.Vector(head_a,15,34,56)*a_mass)
head_a.add_force(soya.Vector(scene,-150,1,-505)*a_mass)
head_a.set_xyz(20,-5,0)
head_a.turn_y(30)
head_b.x = 1.0
head_b.add_force(Vector(head_b,0,0,-1368),
    Vector(head_b,0.0001,0.0002,0.0003))
head_c.set_xyz(-4.5,2.7,-77)
head_c.turn_y(180)
head_d.set_xyz(1,-50,-60)
head_d.turn_x(90)
head_d.add_force(soya.Vector(scene,0,350,-40)*
    d_m,soya.Vector(head_d,0.001,0.0015,0.002))
```

Wir fügen außerdem noch Gravitation hinzu:

```
scene.gravity = soya.Vector(scene,0.0005,-0.03,2)
```

Danach wird ODE bekanntgegeben, dass es sich bei den Köpfen der Raupen um Kugeln mit dem Durchmesser 1.2 handelt:

```
for head in (head_a,head_b,head_c,head_d,):
    soya.GeomSphere(head,1.2)
```

Das Ganze sieht dann wie folgt aus:

8.3 Turm von Blöcken

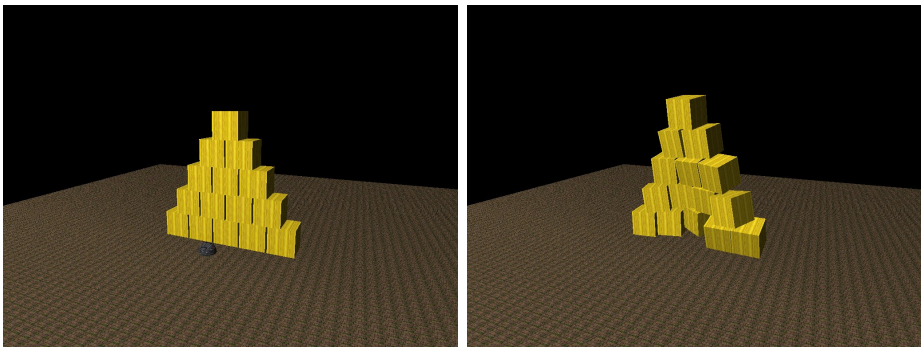
Im folgenden Beispiel soll simuliert werden, wie ein Turm aus Blöcken einstürzt. Dazu modellieren wir zuerst den Turm auf einem Boden und bewegen dann eine Kugel hinein. Durch die ODE Unterstützung in Soya funktioniert der Rest von alleine, sobald wir auf die Kugel eine Kraft wirken lassen, sodass sie sich in den Turm bewegt. Der relevante Code dazu sieht wie folgt aus:

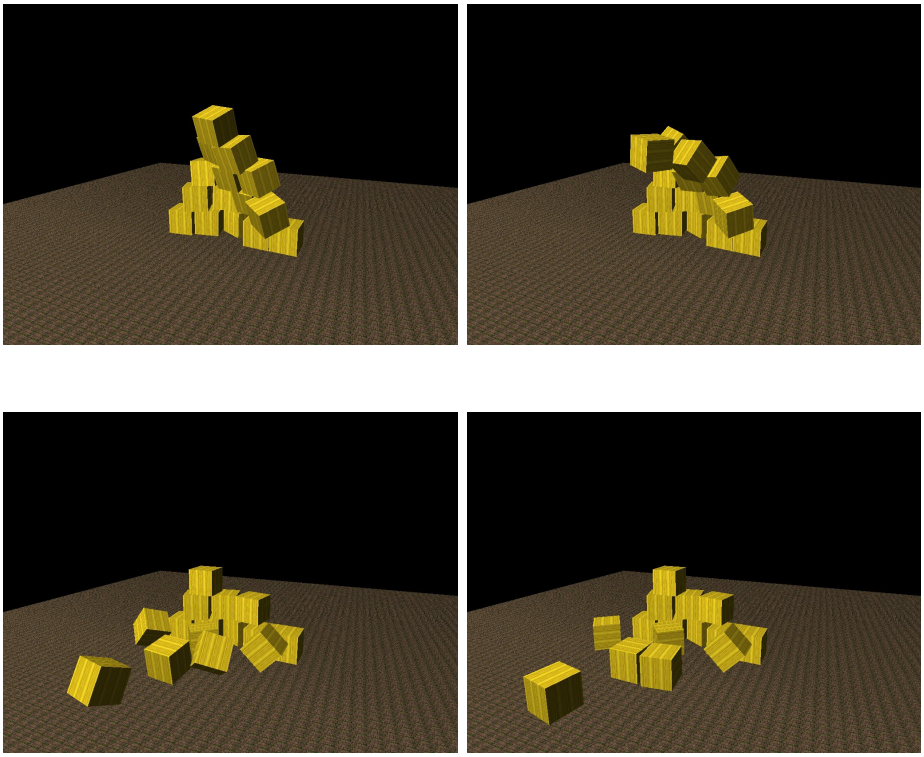
Listing 8.7: [ode-collision-9-box.py]Einstürzender Turm

```
ground = soya.Material(soya.Image.get("block2.png"))
metal  = soya.Material(soya.Image.get("metal1.png"))
cube_mat    = soya.Material(soya.Image.get("chaume.png"))

m_ball = soya.sphere.Sphere(None, metal).shapify()
m_cube = soya.cube.Cube(None, cube_mat, size=3).shapify()
m_ground = soya.cube.Cube(None, ground, size=78).shapify()

ground = soya.Body(scene,m_ground)
ball   = soya.Body(scene,m_ball)

cubes = []
for i in xrange(15):
    cubes.append(soya.Body(scene,m_cube))

ball_density = 50
ground.pushable = False
ground.gravity_mode = False
```

```
ground.mass       = soya.SphericalMass(1)
ball.mass         =soya.SphericalMass(ball_density)
for cube in cubes:
    cube.mass =soya.BoxedMass(0.01, 3, 3, 3)
scene.gravity = soya.Vector(scene,0,-9.8,0)
ball.bounciness = 1
soya.GeomSphere(ball)
for cube in cubes:
    soya.GeomBox(cube,(3,3,3))
soya.GeomBox(ground,(78,78,78))
ground.y-= 39
ball.z    = 10
ball.y    = 0.6
ball.x    = -1
cubes[0].set_xyz(    0,14.0,0)
cubes[1].set_xyz(-1.6, 10.90,0)
cubes[2].set_xyz( 1.6, 10.90,0)
cubes[3].set_xyz(-3.2, 7.80,0)
cubes[4].set_xyz(    0, 7.80,0)
cubes[5].set_xyz( 3.2, 7.80,0)
cubes[6].set_xyz(-4.8, 4.70,0)
cubes[7].set_xyz(-1.6, 4.70,0)
cubes[8].set_xyz( 1.6, 4.70,0)
cubes[9].set_xyz( 4.8, 4.70,0)
cubes[10].set_xyz(-6.4,1.60,0)
cubes[11].set_xyz(-3.2,1.60,0)
cubes[12].set_xyz(    0,1.60,0)
cubes[13].set_xyz( 3.2,1.60,0)
cubes[14].set_xyz( 6.4,1.60,0)
ball.add_force(soya.Vector(scene,ball_density*-50,0,ball_density*-2500))
```

Es werden zuerst die Texturen, also das Material geladen. Damit werden dann die drei Modelle erzeugt, also Kugel, Block und Boden. Von der Kugel und dem Boden erzeugen wir je ein Objekt, von den Blöcken insgesamt 15. Für den Boden setzen wir `pushable` auf false, da er sich durch einen Zusammenstoß mit einem Block nicht bewegen soll. Außerdem soll er durch die Gravitation nicht angezogen werden. Daher wird noch `gravity_mode` auf false gesetzt. Jedem Objekt wird eine Masse und eine geometrische Form zugewiesen. Zuletzt werden alle Objekte noch platziert und auf die Kugel eine Kraft ausgeübt. Die Simulation läuft nun so ab:





8.4 Terrain

Soya3d unterstützt auch Terrains. Wir haben ein solches bereits im Kapitel grafische Effekte (siehe 3.1) gesehen. Hier wird die Implementation betrachtet. Zudem wird hier gezeigt wie man mit einem vorgegebenen Terrain andere Objekte rollen lassen kann ohne sich dabei selbst um die Physik kümmern zu müssen. Um ein Terrain zu erstellen, benötigt man ein 2D-Bild. Die Helligkeit der Punkte gibt dabei die Höhe des Terrains an. Für unser Beispiel verwenden wir folgendes Bild:

Um aus diesen Bild ein Terrain zu machen, benötigen wir ein `soya.Terrain` Objekt. Diesem wird das oben gezeigte Bild zugewiesen:

Listing 8.8: [my-terrain-step1.py]Terrain

```
terrain = soya.Terrain(scene)
terrain.from_image(soya.Image.get("map3.png"))
```

Weiters soll das Terrain stärker gewellt sein, d.h. die maximale Höhe soll größer sein. Normalerweise reicht die Höhe von 0.0 bis 1.0. Man kann diesen Wert jedoch noch multiplizieren:

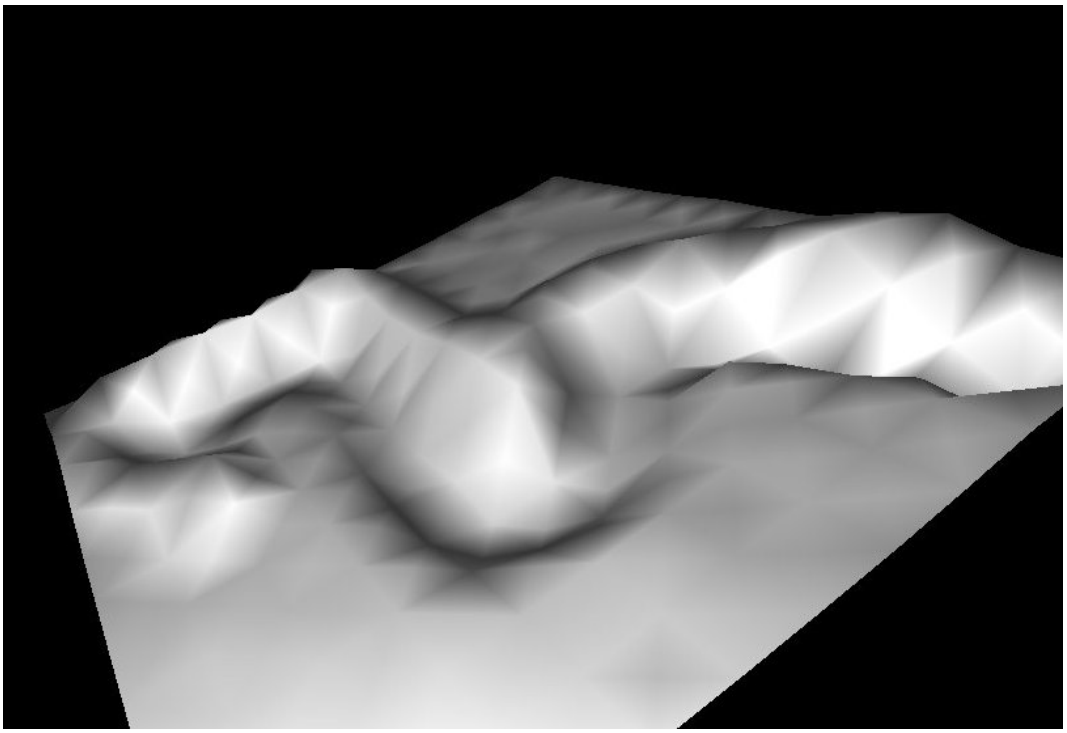
```
terrain_height =  100.
terrain.multiply_height(terrain_height)
```

Zusätzlich muss man noch einige Faktoren setzen:

```
terrain.scale_factor    = 1.0
terrain.split_factor    = 2.0
```

```
terrain.geom = True
```

Fügt man noch Licht und Kamera der Szene hinzu bekommt ein Bild wie folgendes:

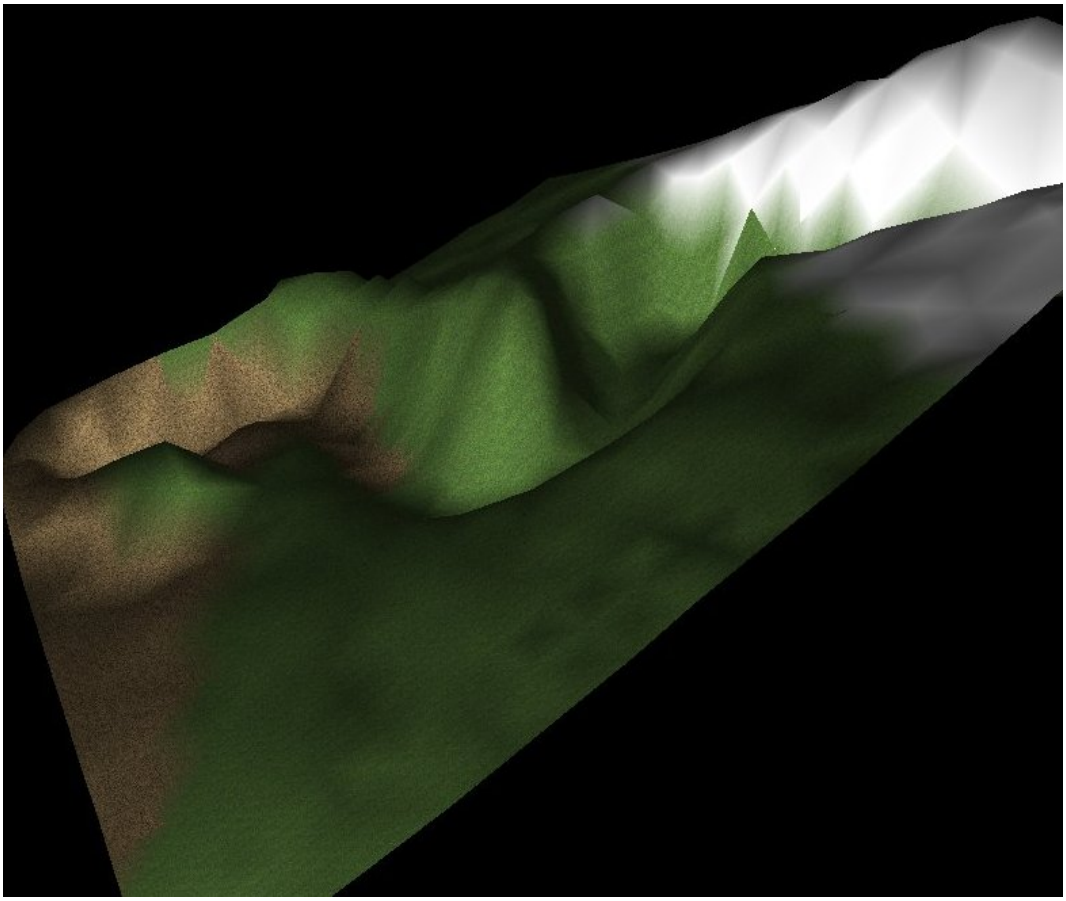


Dieses Bild sieht noch etwas fad aus. Um es interessanter zu gestallten können wir noch eine Textur anbringen. Man kann für unterschiedliche Höhen andere Texturen verwenden. Im Folgenden verwenden für sehr tief gelegene Stellen eine Textur, die nach Erde aussieht. In der Mitte verwenden wir eine die grün ist und den Eindruck einer Graslandschaft erweckt. Die ganz hohen Teile lassen wir weiß, so dass die Spitzen beschneit aussehen. Dies wird durch Hinzufügen von folgendem Code realisiert:

Listing 8.9: [my-terrain-step2.py]Texturen auf Terrain

```
ground = soya.Material(soya.Image.get("ground.png"))
grass  = soya.Material(soya.Image.get("grass.png"))
terrain.set_material_layer(ground, 0.0, 5*terrain_height/12.)
terrain.set_material_layer(grass, 5*terrain_height/12.0, 7*terrain_height/12.)
```

Hierbei wird bei der Höhe von 0 bis $\frac{5}{12}$ die Textur `ground` angewendet und von $\frac{5}{12}$ bis $\frac{7}{12}$ die Textur `grass`. Von $\frac{7}{12}$ bis 1 bleibt das Terrain weiß. Das ganze sieht wie folgt aus:



Bis jetzt haben wir lediglich das Terrain gezeichnet, nun wollen wir ein Objekt darauf bewegen lassen. Also erzeugen wir noch ein Raupen Kopf und lassen in auf das Terrain fallen. Dieser bewegt sich dann entsprechend der Kontur und den physikalischen Gesetzen:

Listing 8.10: [my-terrain-step3.py]Rollende Objekte

```
class Head(soya.Body):
    model = soya.Model.get("caterpillar_head")
    def __init__(self,parent):
        soya.Body.__init__(self,parent, self.model)
        self.mass = soya.SphericalMass(12,1,"total_mass")
        self.geom = soya.GeomSphere(self,1.2)

head_a = Head(scene)
head_a.set_xyz(77, terrain_height, 20)
```

8.5 Joints

Joints sind Gelenke, also Punkte an den ein Objekt fixiert ist. Diese werden von ODE unterstützt und sind auch in Soya implementiert. Die Funktionsweise wollen wir an einem

Schwert demonstrieren welches an einem Kugelgelenk fixiert ist und auf dem unterschiedliche Kräfte wirken. Das Schwert dreht sich dadurch um den Gelenkspunkt und behält diese Position bei.

Listing 8.11: [ode-joint.py]Kugelgelenke

```
blade = soya.BoxedMass(0.00005,50,5,1)
pommeau = soya.SphericalMass(50,0.5)
pommeau.translate((25,0,0))
sword.mass = blade+pommeau
joint = soya.BallJoint(sword)

sword.add_force(v(10,200,0),v(25,0.001,0.002))
sword.add_force(v(0,60,0),v(0,25,0.5))
sword.add_force(v(0,0,-2303*5),v(-25,0.0001,0.0001))
```

In diesem Beispiel zeichnen wir ein Schwert und setzen seine Masse. Außerdem definieren wir ein Gelenk (BallJoint). Danach lassen wir noch ein Kraft darauf wirken, sodass sich das Schwert um den Gelenkspunkt dreht.

9 Pudding

Pudding ist der Teil innerhalb Soya3D, der dazu dient, Objekte zu zeichnen die oberhalb der eigentlichen grafischen Animation liegen. Dazu gehören z.B. Texte, Anzeigebalken, Buttons oder andere Steuerelemente. Pudding ist in einem eigenen Modul. Daher benötigt man zusätzlich ein `import`:

Listing 9.1: [pudding-1.py]Einfacher Text

```
import soya
import soya.pudding as pudding

soya.init()
pudding.init()
```

Nun lassen sich alle Elemente im Pudding-package über `pudding` ansprechen, anstatt über `soya.pudding`.

Damit die Elemente in pudding auch angezeigt werden, muss man das root-Widget in Soya3d setzen. Dieses benötigen wir allerdings auch für die Kamera. Das pudding RootWidget hat jedoch auch die Möglichkeit zusätzliche Elemente diesem hinzuzufügen. Dies können wir folgendermaßen nutzen:

```
soya.set_root_widget(pudding.core.RootWidget())
soya.root_widget.add_child(camera)
```

9.1 Text

Um Text anzuzeigen benötigt man das SimpleLabel-Widget:

```
text = pudding.control.SimpleLabel(soya.root_widget,
    label = "Hello World!", autosize = True)
```

Damit ist eigentlich auch schon alles erledigt. In der Hauptschleife wird damit über `soya.root_widget` erkannt, dass ein Simple-Label existiert und entsprechend angezeigt. Der Parameter `autosize` gibt an, dass die Größe des Widgets anhand der Größe und Länge des Textes bestimmt wird. Alternativ kann man sie auch mit `width` und `height` setzen.

Man kann außerdem die Schrift verändern. Dazu überprüft man am besten zuerst welche Schriften auf dem Zielsystem installiert sind:

Listing 9.2: [get-fonts.py]Vefügbare Schriften auflisten

```
import soya.pudding as pudding
pudding.sysfont.get_fonts()
```

Nun können wir eine Schrift auswählen und auf unseren Text anwenden:

Listing 9.3: [pudding-font.py]Schrift auswählen

```
myfont = soya.Font(pudding.sysfont.SysFont('couriernew'), 15, 15 )
text = pudding.control.SimpleLabel(soya.root_widget,
    label = "Hello World!", font=myfont, autosize = True)
```

Es ist außerdem möglich den Text zu platzieren. Dazu übergibt man die Parameter `left` und `top`. Der Aufruf um so ein Label zu setzen sieht dann wie folgt aus:

Listing 9.4: [pudding-place.py]Widgets platzieren

```
text = pudding.control.SimpleLabel(soya.root_widget, left=100,
    top=100, label = "Hello World!", font=myfont, autosize = True)
```

9.2 Buttons

Neben Text gibt es auch die Möglichkeit Buttons zu platzieren. Diese können Beispielsweise die Funktion eines Quit-Buttons haben, man kann sie verwenden um ein Menu zu implementieren oder als zusätzliche Steuerungsmöglichkeit.

Listing 9.5: [pudding-btn.py]Button

```
button = pudding.control.Button(soya.root_widget, 'Quit',
    left = 10, width = 50, height = 40)
button.on_click = sys.exit
```

Hiermit wird ein Button mit dem Namen Quit erzeugt und die Postion, sowie die Größe angegeben. Weiters wird in der nächsten Zeile die Funktion festgelegt die bei einem Klick-Event aufgerufen werden soll.

9.3 Logo

Man kann auch ein Bild platzieren, das seine Position unabhängig von der Bewegung der Kamera beibehält. Dies geht mit dem Pudding-Widget Logo:

Listing 9.6: [pudding-logo.py]Bild

```
logo = pudding.control.Logo(soya.root_widget, 'little-dunk.png')
```

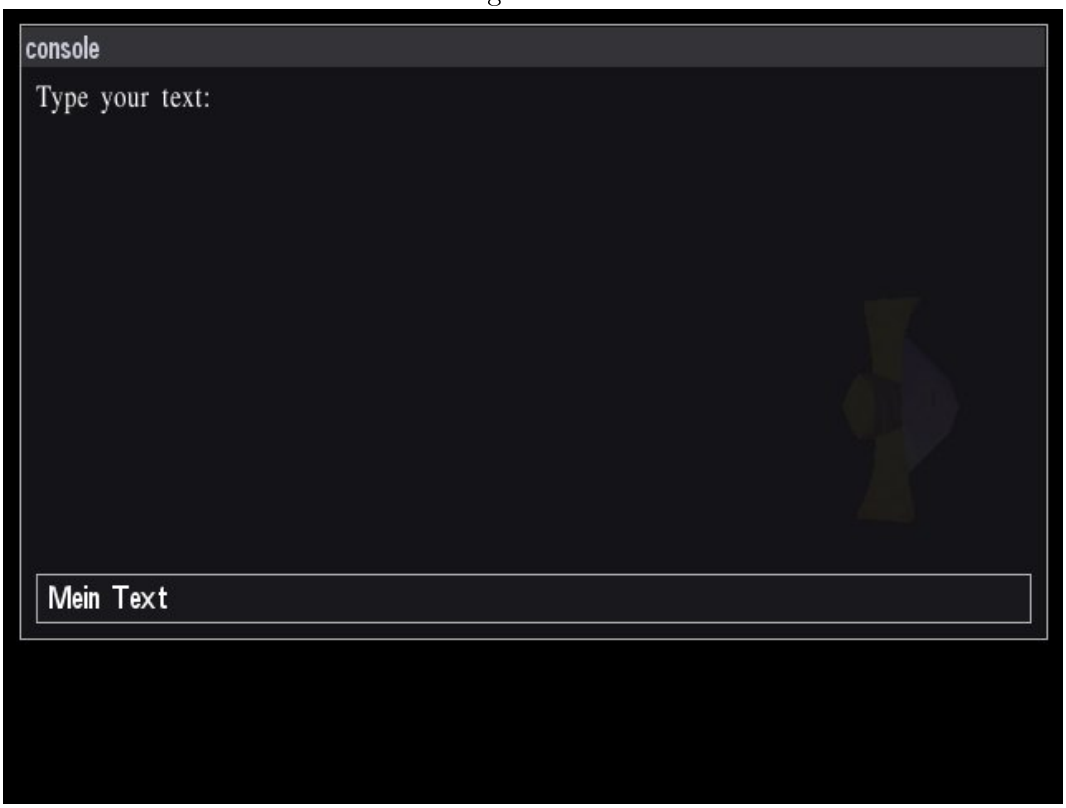
9.4 Konsole

Pudding hat auch ein Konsolen-Widget. Diese kann dazu verwendet werden um das Spiel zu steuern oder auch einfach als Chat-Fenster. Ein Konsolen-Widget kann wie folgt verwendet werden:

Listing 9.7: [pudding-console.py]Konsole

```
msg = "Type your text: "
console = pudding.control.Console(soya.root_widget, left=10, top=30, initial = msg)
console.right=10
console.bottom=10
console.output.font = myfont
```

Damit entsteht ein Fenster das wie folgt aussieht:



9.5 Container

Container sind Elemente, die weitere Widgets enthalten können. So ist es möglich die Widgets entsprechend zu gruppieren und somit das korrekte Platzieren zu ermöglichen. Wir wollen nun einen Container verwenden, um eine Leiste von Buttons zu erstellen. Ohne einen Container wäre dies recht mühsam, da für jeden Button manuell die exakte Position angegeben werden müsste. Der Container übernimmt diese Arbeit. Der Code dafür sieht so aus:

Listing 9.8: [pudding-buttonbar-1.py]Reihe von Buttons

```
w = pudding.core.RootWidget(width = 1024,height = 768)
button_bar = pudding.container.HorizontalContainer( w, left = 10,
    width= 164, height=64)
button_bar.set_pos_bottom_right(bottom = 10)
d = button_bar.add_child(pudding.control.Button(label = 'Button1'),
    pudding.EXPAND_BOTH)
f = button_bar.add_child(pudding.control.Button(label = 'Button2'),
    pudding.EXPAND_BOTH)
```

Zuerst wird eine `button_bar` als horizontaler Container erzeugt. D.h. die Elemente in diesem Container werden horizontal angeordnet. Die `button_bar` selbst wird 10 Pixel

vom unteren Rand des Fensters platziert. Dann werden zwei Buttons dem Container hinzugefügt und angegeben, dass sie in beide Richtungen expandieren dürfen. Es natürlich ist wie bei anderen GUI-Toolkits möglich, Container ineinander zu schachteln.

10 Blender

Blender[11] ist ein Open Source Tool für 3D-Modellierung, Animation und Rendering. Die Modells aus Blender können in Soya3D verwendet werden. Daher ist ein Grundwissen darüber hilfreich. Es kann allerdings hier nur eine kurze Einführung gegeben werden, um den Rahmen dieser Arbeit nicht zu sprengen. Es gibt jedoch Literaturangaben zu diesem Thema.

Da wir 3D-Modellierung betreiben, ist es notwendig ein Objekt von mehreren Seiten betrachten zu können. Blender bietet dazu mehrere Ansichten. Um eine neue Ansicht hinzuzufügen bewegt man die Maus über den Rand einer bestehenden bis das Maussymbol auf das Verschieben-Symbol umspringt, drückt die rechte Maustaste und wählt "split area". Für jede Area wählt kann man jetzt mit den Tasten des Numpads die Gewünschte Ansicht auswählen.

10.1 Drahtgittermodell

Der nächste Schritt ist das Hinzufügen von Objekten. Dazu wird mit Space die Toolbox aufgerufen. Dort kann unter Add->Mesh->Cube ein Würfel hinzugefügt werden. Objekte werden mit der rechten Maustaste ausgewählt und können mit "g" verschoben, mit "s" skaliert und mit "r" rotiert werden. Alternativ kann auch Shift + mittlere Maustaste zum verschieben, Ctrl + mittlere Maustaste zum Skalieren verwendet werden.

Blender hat mehrere Modelle. Das erste ist der Objekt Modus, um z.B. Objekte zu bewegen. Um ein Drahtgittermodell zu bearbeiten, gibt es den Edit Modus. Der dritte Modus ist der Face Modus der dazu verwendet wird ein um eine Texture anzuwenden. Der Modus kann auf der Menuleiste unten bei der Ansicht ausgewählt werden.

10.2 Blender und Soya

Blender hat eine andere Konvention für Achsenbezeichnungen. Bei Blender weist die X-Achse nach rechts, die Y-Achse nach vorne und die Z-Achse nach oben. Bei Soya3D ist jedoch die Y-Achse nach oben und die Z-Achse nach hinten gerichtet. Dies wird jedoch durch den Exporter berücksichtigt, welcher das Modell entsprechend rotiert.

Außerdem ist zu beachten, dass nur Dreiecke und Quadrate durch Soya3D berücksichtigt werden. Komplexere Faces, Linien oder Punkte werden zwar durch Blender, aber nicht durch Soya3D unterstützt.

10.3 Beleuchtung

Mit den Einstellungen "Set Smooth" bzw. "Set Solid" kann man einstellen, ob die Beleuchtung für ein Objekt weich verlaufen soll oder ob man harte Kanten sehen können soll. Manche Objekte sind von Natur aus weich, zum Beispiel eine Kugel. Bei anderen Objekten hingegen sollen die Kanten sichtbar sein. Wenn der Winkel zwischen zwei Faces größer als 80 Grad ist wird automatisch solid gewählt (Man kann dies jedoch über den Parameter `max_face_angle` verändern).

10.4 Texturen

Texturen sind gewöhnliche 2D-Bilder und können z.b. in einem gewöhnlichen 2D Bildbearbeitungsprogramm wie Gimp[15] hergestellt werden. Entweder verwendet man ein Foto dafür oder wendet kompliziertere Methoden an[1]. Texturen sollten entweder JPEG oder PNG-Bilder sein und die Endung ".jpeg" oder ".png" haben. Diese werden in das Verzeichnis <data>/images (siehe Kapitel 5.1) abgelegt.

Soya3D unterstützt RGB und RGBA-Bilder. Die Dimensionen der Bilder müssen eine Potenz von 2 sein, aber es ist nicht notwendig, dass das Bild quadratisch ist. Texturen mit Alpha-Channel, also Bilder die Transparenz unterstützen, werden langsamer verarbeitet als Bilder ohne Alpha-Channel. Manche Bildverarbeitungsprogramme fügen jedoch häufig einen unerwünschten Alpha-Channel ein. Dieser sollte entfernt werden.

In Blender ist der Name für eine Textur auf 19 Zeichen limitiert. Der Exporter nimmt an, dass dies der Name ist für die dazugehörige Bilddatei. Daher sollte man lange Namen für die Texturen vermeiden.

Um eine Textur auf ein Objekt anzuwenden muss man zuerst, wie bereits erwähnt in den Face Mode wechseln. Dann wählt man im UV/image Editor Fenster in der unteren Leiste Image->Open und öffnet die Textur. Dann legt man noch die Punkte des Dreicks oder Quadrats über die Textur.

Eine andere Möglichkeit ein Objekt einzufärben sind Face Colors. Diese werden jedoch als Punktfarben exportiert, da Soya3D keine Face Colors unterstützt.

10.5 Armature

Bei animierten Objekten verwendet Blender das Skelett eines Modells. Dieses wird in Blender als Armature bezeichnet. Diese werden dann mit einzelnen Punkten verlinkt. Um einen Knochen zu erzeugen drückt man zuerst die Leertaste und wählt dann Add->Armature im Menü aus. Nun kann man mit Leertaste gefolgt von Add->Bone die Knochen hinzufügen und diese mit der Maus verschieben bzw. skalieren. Danach werden die Punkte mit den Knochen verlinkt. Dazu wird zuerst das Modell und dann die Armature ausgewählt. Dann klickt man auf Object->Parent->Make parent->Armature->Create und es werden Punktgruppen zu den Knochen automatisch erzeugt, die manuell bearbeitet werden können.

Um das Skelett zu bewegen benötigt man nun den Action Editor. Dort kann man für die einzelnen Knochen Aktionen hinzufügen. Wie man Bewegungen animiert steht in [19].

10.6 Auto Exporter

Der einfachste Weg ein Blender Modell nach Soya3D zu exportieren ist über den Auto Exporter, siehe auch Kapitel 5.1. Jedes Mal wenn Soya3D ein Modell lädt, sucht es zuerst ein Soya3D-Modell. Wenn dieses nicht vorhanden ist nach einem Blender Modell und konvertiert dies zu einem Soya3D-Modell. Daher kann man ein winziges Python Programm schreiben, das die Konvertierung durchnimmt. Dazu kopiert man das Blender-Modell nach `<data>/blender/model.blend`. Danach führt man folgende Zeilen aus:

Listing 10.1: [blend2soya.py]Auto Exporter

```
soya.Model.get("model")
soya.AnimatedModel.get("model")
```

Damit werden sowohl animierten als auch nicht animierte Modelle exportiert. Soya3D ruft dabei kurzfristig Blender auf, schließt es allerdings sobald es fertig ist. Es werden nicht alle Features von Blender exportiert, da Blender wesentlich mehr kann, als von Soya3D benötigt, allerdings werden folgende Elemente unterstützt:

- Drahtgittermodell
- Face UV image
- Vertex UV texture coordinates
- Face twoside
- Smooth und solid lighting
- Face colors

Für nicht animierte Modelle:

- SubSurf

Nur für animierte Modelle:

- Armature
- Actions

10.7 Soya3D spezifische Attribute

Soya3D kennt einige Attribute, die von Blender nicht direkt unterstützt werden. Diese können über einen Text Buffer Soya3D mitgeteilt werden. Folgende Parameter sind bekannt:

config_file=file Über diese Option kann man eine Datei angeben, die wie der Parameter Buffer gelesen wird. Wer die Paramater nicht in einen Text Buffer schreiben will kann diese Option verwenden und die restlichen Parameter in die Textdatei auslagern.

config_text=blender Hier kann der Name eines weiteren Text Buffers angegeben der ebenfalls gelesen werden soll. Ohne diese Option wird nur der Buffer soya_params gelesen.

text buffer name also read the Blender text buffer of the given name.

scale=2.0 Diese Option skaliert das Objekt.

shadow=1 Standardmäßig sind Schatten deaktiviert (shadow=0). Mit dieser Option (shadow=1) kann man sie einschalten.

cellshading=1 Hiermit kann man *Cel Shading* auf dem Soya3D Modell aktivieren. Standardmäßig ist es deaktiviert.

cellshading_shader="filename" Hiermit kann man den Namen des Materials angeben, das vom Shader verwendet wird. Voreingestellt ist soya.SHADER_DEFAULT-_MATERIAL. Es werden nur die Texturen von dem Material verwendet, welche eine ein Pixel weite Alpha Textur sein soll. Diese Texture wird dann über die normal Textur des Models gelegt, mit den obersten Pixeln über den dunkleren des Models und den untersten über die hellsten Teile.

cellshading_outline_width=1.0 Wird nur verwendet, wenn Cel Shading aktiviert ist. Voreingestellt ist 0.0, was bedeutet das kein Umrandung hinzugefügt wird. Ansonsten gibt der Wert die dicke der Umrandung an.

cellshading_outline_color=red,green,blue,alpha Die Farbe der Umrandung bei Cel Shading. Wird nur verwendet wenn Cel Shading aktiviert ist und die Umrandungsdicke größer als 0.0 ist. Voreingestellt ist schwarz.

cellshading_outline_attenuation=0.3 Die Farbe der Umrandung ist nicht überall gleich sondern wird nach außen hin heller. Um wieviel heller es werden soll kann mit dieser Option eingestellt werden. Voreingestellt ist 0.3.

animation=blender Hiermit kann man eine Blender Aktion als aktuell setzen bevor sie exportiert wird.

animation_time=3.0 Hiermit wird die Framerate gesetzt. Diese Option wird in der Regel gemeinsam mit der animation Option verwendet.

max_face_angle=80.0 Hiermit kann man den maximalen Winkel angeben, bei dem gerade noch die Faces weich beleuchtet werden. Ist der Winkel größer als hier angegeben, so werden die Faces nicht mehr weich verlaufend dargestellt auch wenn sie als solche von Blender gekennzeichnet sind. In anderen 3D Engines und auch in Modellierungsprogrammen wie Blender, muss man Punkte verdoppeln um die weichen Übergänge zu verhindern. Im Gegensatz dazu verwendet Soya3D den Winkel und übernimmt somit die Arbeit. Voreingestellt ist 80.0. Um diese Fähigkeit abzustellen kann man diese Option auf 360.0 setzen.

keep_points_and_lines=1 Wenn diese Option auf 1 gesetzt ist bleiben Linien und Punkte im Modell. Normalerweise verwirft Soya3D diese und behält nur Drei- und Vierecke. Voreingestellt ist 0.

material_oldname=newname Mit dieser Option kann man ein Material umbenennen.

11 Raypicking

Raypicking ist ein Verfahren um den ersten Schnittpunkt mit einem Objekt zu bestimmen.

Soya3D ermöglicht 2 verschiedene Varianten. Das direkte Schneiden mit Raypicking ergibt ein genaues Ergebnis und verhindert Treffer die gar nicht stattgefunden haben, hat aber das Problem, dass es performancemäßig nur für wenige Objekte in einer Szene einsetzbar ist.

Deshalb wurde soya tracing eingeführt, welches ein neues collisions detection system für Soya3D, inspiriert von Quake3, ist. Es besticht durch seine schnellen und einfach handzuhabenden Tracingmöglichkeiten.

11.1 Laser

Der Laser in Soya3D ist ein roter Strahl der von einem Startpunkt bis zu einem Zielpunkt geht. Er kann somit verwendet werden, um eine Linie zu zeichnen.

```
laser = soya.laser.Laser(scene)
```

Obiger Code erzeugt bereits einen roten Strich von Mittelpunkt eine Längeneinheit in die Höhe. Wir können die üblichen Operationen verwenden um ihn zu rotieren oder woanders hin zu setzen, z.b.:

```
laser = soya.laser.Laser(scene)
laser.y = -2.5
laser.rotate_y(20)
```

Im Kontext von Raypicking sehr interessant ist aber, dass er stoppt oder reflektiert wenn er Objekte trifft. Dies kann durch die Option `reflect` im Konstruktor gesteuert werden. Die Farbe kann durch `color` geändert werden.

Wir wollen den Laser nun auch mit der Maus steuern. Dazu leiten wir MouseLaser von Laser ab:

```
class MouseLaser(soya.laser.Laser):
    def begin_round(self):
        soya.laser.Laser.begin_round(self)
        for event in soya.process_event():
```

Bei Bewegung der Maus passiert das entscheidende, wir schauen zu den Punkt hin wo der Laser hingehen soll. Dafür verwenden wir die Mauskoordinaten mit dem z-Wert 0.0.

```
            if event[0] == soya.sdlconst.MOUSEMOTION:
                mouse = soya.Point(
                    scene,
                    (float(event[1])/camera.get_screen_width()-0.5)* 4.0,
```

```
                    (float(event[2])/camera.get_screen_height()-0.5)*-4.0,
                    0.0,
                    )
                self.look_at(mouse)
laser = MouseLaser(scene, reflect = 0)
```

11.2 Reflexionen

Wir wollen den Laser nun verwenden um Reflexionen an Körpern zu visualisieren. Dies kann mit der bis jetzt nicht benutzen Option `reflect` geschehen.

11.2.1 bei Würfel

Das erste Beispiel zeigt drei rotierende Würfel. Wir wollen einen Laserstrahl setzen der bei den Würfeln richtig reflektiert wird.

Listing 11.1: [raypicking-1.py]Reflexionen bei Würfel

```
soya.cursor_set_visible(0)
```

Damit kann der Mauscursor ausgeblendet werden. Dies wird hier gemacht, da der Laserstrahl bereits als Cursor wirkt.

```
class MouseLaser(soya.laser.Laser):
    def begin_round(self):
        soya.laser.Laser.begin_round(self)
```

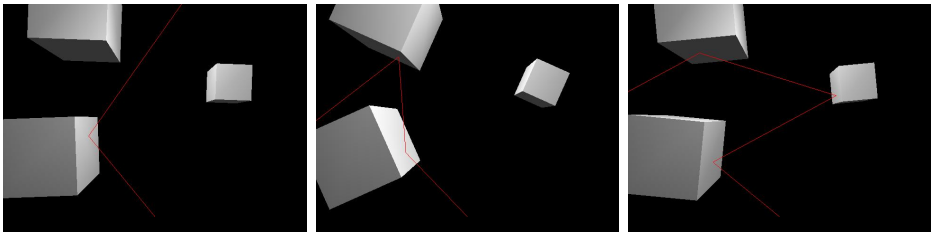
Wir erschaffen uns eine neue Klasse MouseLaser, das ist ein Laser der durch die Bewegungen der Maus gesteuert wird:

```
        for event in soya.process_event():
            if event[0] == soya.sdlconst.MOUSEMOTION:
                mouse = soya.Point(
                    scene,
                    (float(event[1])/camera.get_screen_width()-0.5) * 4.0,
                    (float(event[2])/camera.get_screen_height()-0.5) *-4.0,
                    0.0,
                    )
                self.look_at(mouse)
```

Die besondere Eigenschaft des MausLaser ist dass er immer auf die Koordinaten der Maus schaut. Um das zu erreichen wird die x-Koordinate durch die Auflösung der Bildschirmbreite dividiert und mit Faktoren so gewichtet, dass der Laser immer nach oben schaut und sich mit einer angenehmen Geschwindigkeit ändern lässt.

```
laser = MouseLaser(scene, reflect = 1)
laser.y = -2.5
scene.x = 1.0
soya.MainLoop(scene).main_loop()
```

Diesen Laser instanzieren wir jetzt, allerdings mit der Option `reflect` auf 1 gesetzt.



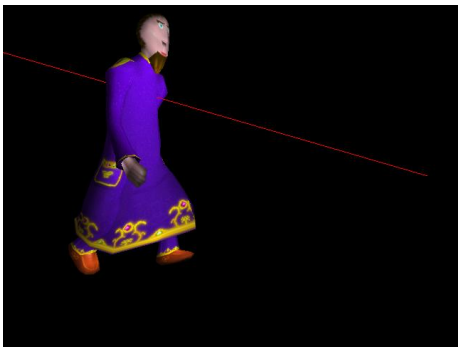
11.2.2 bei animierten Charakter

Bei einem animierten Charakter gehen wir gleich vor. Es gibt allerdings ein paar Feinheiten zu beachten.

Listing 11.2: [raypicking-3.py]Reflexionen bei animierten Charakter

```
import sys, os, os.path, soya, soya.cube, soya.laser, soya.sdlconst
soya.init()
soya.path.append(os.path.join(os.path.dirname(sys.argv[0]), "data"))
scene = soya.World()
sorcerer_model = soya.AnimatedModel.get("balazar")
sorcerer = soya.Body(scene, sorcerer_model)
sorcerer.rotate_y(-120.0)
sorcerer.animate_blend_cycle("marche")
sorcerer.set_xyz(-1.0, -1.0, 0.0)
sorcerer.solid = 1
```

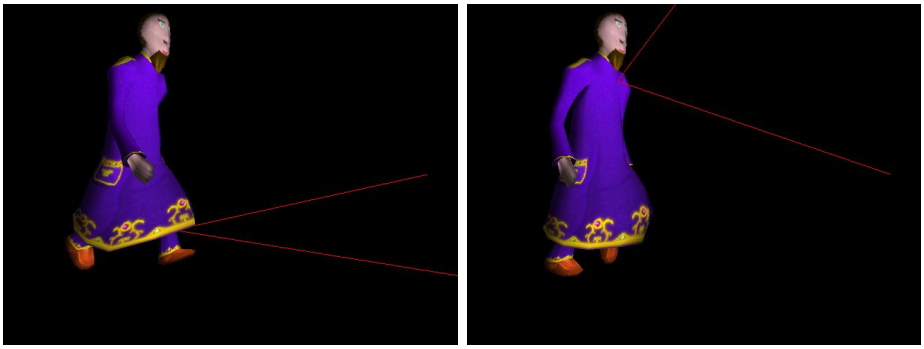
Neben dem Laden der Modelle und Starten der Animation muss hier beachtet werden, dass das Modell als `solid` deklariert wird. Anderenfalls geht der Strahl einfach durch ihn durch, das heißt Raypicking findet nicht statt, wie folgende Grafik verdeutlicht:



Der MouseLaser wird wieder gleich implementiert, diesesmal allerdings auf einer anderen Position gesetzt.

```
laser = MouseLaser(scene, reflect = 1)
laser.x = 2.0
soya.MainLoop(scene).main_loop()
```

Wird das Modell richtig als solid markiert, so erhalten wir das erwünschtes Ergebnis:



11.3 Standard

Die Standardvariante von Raypicking, die auch vom Laser verwendet wird ist `raypick`. Der Laser wird durch folgende Zeilen implementiert:

Listing 11.3: [laser.py]Implementierung des Lasers

```
while direc and (i < 50):
    i = i + 1
    impact = raypicker.raypick(pos, direc, -1.0)
```

Um eine Endlosschleife zu vermeiden, bricht der Laser nach 50 Reflexionen ab. Bei jeder Iteration wird mit `raypicker.raypick` der Schnittpunkt bestimmt. Dabei geht man von der aktuellen Position `pos` aus, und der Einheitsvektor `direc` gibt an, wohin reflektiert wird.

Die eigene Position kann übrigens mit `self.position()` bestimmt werden. Statt `raypicker` wird in eigenen Klassen (welche von World abgeleitet sein müssen) dann `self` verwendet.

```
    if not impact:
        pos    = pos + (direc * 32000.0)
        direc = None
```

Zuerst behandeln wir den Fall, dass es keinen Treffer gibt. Dieser wurde durch `raypick` zurückgegeben. In diesem Fall setzen wir die Position auf einen sehr weit entfernten Punkt und brechen die Berechnung ab.

```
    else:
        pos = impact[0]
```

Ansonsten haben wir einen Treffer und die Postion ist der erste Teil des zurückgegebenen Tupels. `impact.parent` gibt das Coordsys des Objektes zurück.

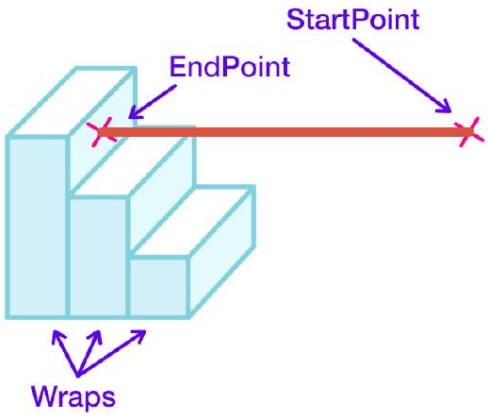
```
        if self.reflect:
            normal = impact[1] % self
            normal.normalize() # changing coordsys can alterate normal size
            normal.set_length(-2.0 * direc.dot_product(normal))
            direc = normal + direc
```

Hier verwenden wir nun die Normale, die als zweiter Teil des Tupels abrufbar ist. Nachdem sie normalisiert wird, d.h. auf Länge 1 gebracht, kann die neue Richtung durch ein `dot_product` berechnet werden (siehe 4.2).

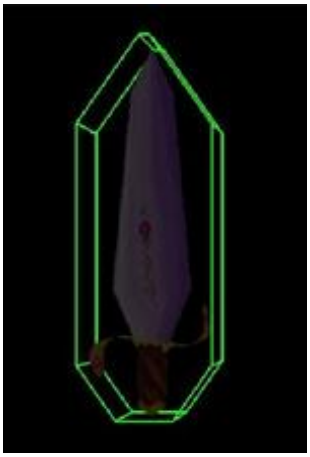
```
else: direc = None
```

Wurde `reflect` (zur Erinnerung: das ist die Option für den Laser welcher das Reflektieren einschaltet) nicht gesetzt, so können wir sofort beenden.

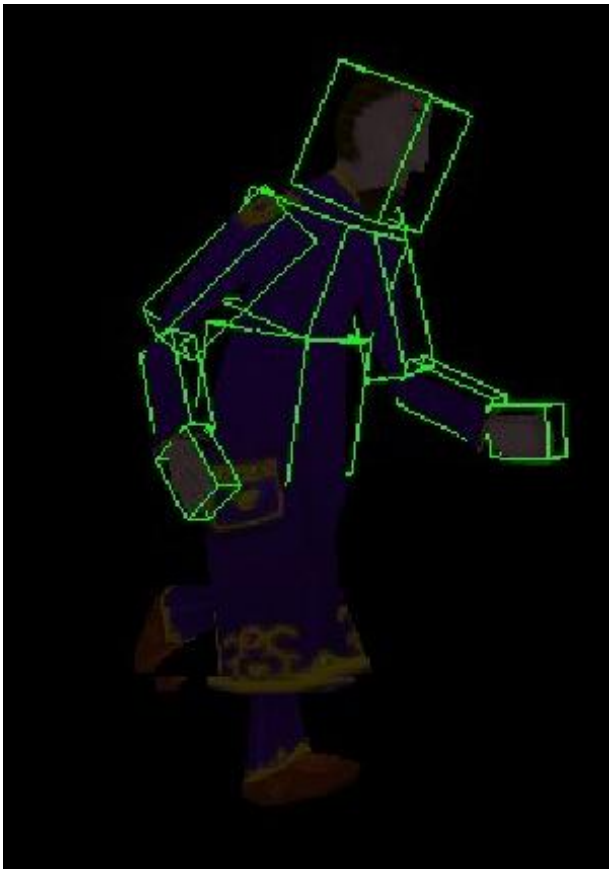
11.4 Wraps



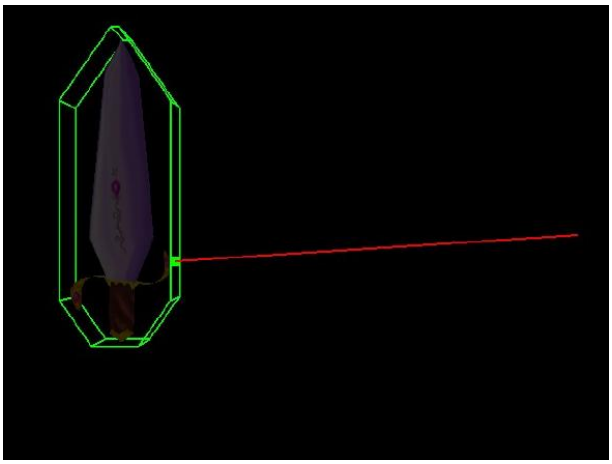
Anstatt direkt mit Objekten zu schneiden kann auch mit Wraps, welche durch die Wrap Klasse repräsentiert werden, geschnitten werden. Wraps werden wie hier gezeigt um ein Objekt gelegt:



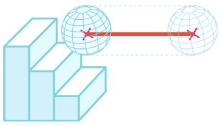
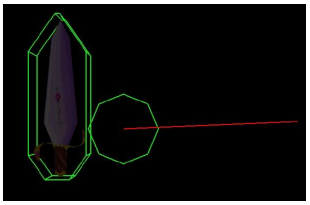
Dies kann auch mit mehreren Wraps bei komplexeren Objekten geschehen. Wraps sind selbstverständlich normalerweise unsichtbar, können aber wie oben für Debuggingzwecke eingeschalten werden.



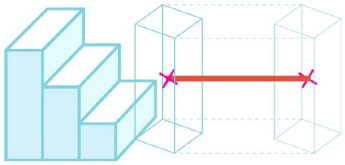
Wraps erlauben dabei klassisch mit Raypicking geschnitten zu werden. Dabei kann man sich vorstellen, dass die Linie entlang gewandert wird bis sie mit einem Wrap schneidet. Es können aber für das Entlangwandern auch Kugeln und Boxen verwendet werden.



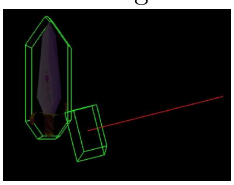
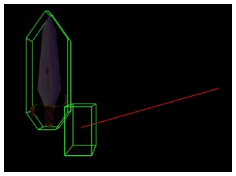
Hier wird eine Kugel der Linie entlang geschoben bis sie ein Objekt trifft. Der Radius der Kugel kann dabei selbst angegeben werden.

Das funktioniert auch mit Boxen.

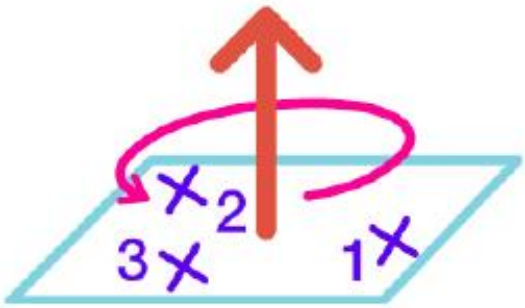


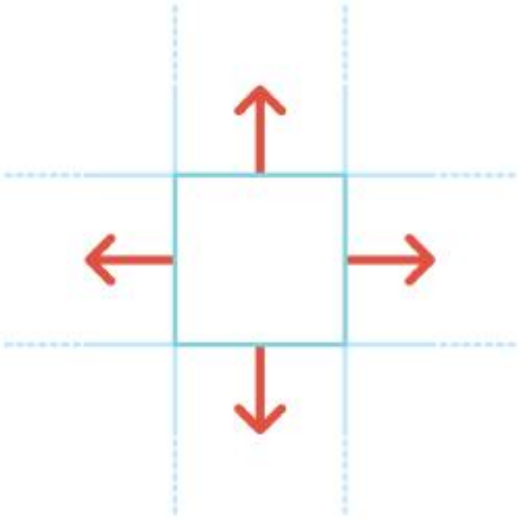
Hier gibt es allerdings verschiedene Orientierungen. Zu einem kann die Box an dem Strahl, aber zum anderem auch an der Szene ausgerichtet sein. Das kann natürlich verschiedene Ergebnisse bringen.

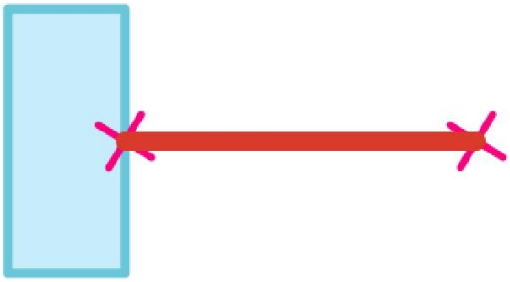
Wraps sind durch Flächen abgegrenzt. Dabei sind einfache Würfel aber auch kompliziertere *konvexe* Körper möglich. Dabei müssen die *Normalvektoren* immer vom Körper wegzeigen.

Sie werden durch eine Liste von Soya Flächen erzeugt.





Wenn der Startpunkt eines Rays ausserhalb von einem Wrap ist, so wird der Ray verfolgt bis er mit einem Wrap schneidet.



Startet hingegen der Ray innerhalb eines Wraps so wird der Schnittpunkt wo der Ray den Wrap verlässt gesucht:

12 Einbinden von Soya3D

Während für Spiele die Tastatur und Maussteuerung für einen Charakter zusammen mit einfachen Menüwidgets genügen haben wissenschaftliche und anderen Anwendungen andere Anforderungen an die GUI. So sollen Dialoge schnell Mithilfe von GUI-Designer erstellt werden und durch qualitativ hochwertige Layout Manager auf verschiedene Auflösungen skalieren und die Elemente selbstständig passend anordnen.

Diese oftmals geforderte Funktionalitäten können nicht direkt von Soya3D erfüllt werden, doch es können problemlos Widgets in anderen Toolkits eine Soya3D Applikation steuern. Dadurch erhält man 2 Fenster, eine grafische 3D Ausgabe und Dialoge eines anderen Toolkits für Eingabe. Es ist aber unter Umständen auch möglich die Soya3D Ausgabe komplett in einer Fremdapplikation einzubetten, dies funktioniert aber nur experimentell.

Das technische Problem dabei ist, dass jedes Toolkit und Engine eine Hauptschleife brauchen in der Ereignisse behandelt werden. Es muss aber sichergestellt sein, dass auch die Hauptschleife von Soya3D alle 30ms aufgerufen wird, sonst könnten wichtige Ereignisse verloren gehen. Bei der eingebetteten Variante muss man zusätzlich noch berücksichtigen wie groß die Soya3D Ausgabe sein darf und Keyevents müssen in einer richtigen Form weitergegeben werden. Die dafür existierenden Ansätze werden hier nicht weiter diskutiert.

12.1 Qt

Qt ist nicht nur ein Toolkit, sondern ein Framework welches von Netzwerk, Dateizugriff bis hin zu Xml ein weites Spektrum von Technologien abdeckt. Bekannt wurde es vor allem durch den exzellenten Layout-Manager und GUI-Designer, welcher es gestattet sehr komplexe Dialoge zu gestalten und wodurch KDE ermöglicht wurde.

Qt ist in C++ geschrieben und kann auch in dieser Sprache verwendet werden, es existieren aber auch Bindings für Python, womit auch eine überwältigende Anzahl von Klassen von dieser Sprache verwendet werden können. Es kommt noch besser, es gibt sogar einen `pyuic` Compiler, mit dessen Hilfe die Xml Dateien vom GUI-Designer zu Python Code, welcher den Dialog generiert, umgewandelt werden können.

Wir betrachten hier ein einfaches Programm welches Soya3D aus einer Qt-Applikation heraus steuern lässt. Anhand ihrer werden grundlegende Konzepte von Qt umrissen, hier kann aber nicht tief genug darauf eingegangen werden, um eigene Qt Programme schreiben zu können.

Listing 12.1: [soya-with-qt.py]Soya aus Qt steuern

```
#!/usr/bin/env python
```

```
import sys, time, os, os.path, soya, soya.widget
from qt import *
```

Neben den üblichen Soya-Importen wird auch Qt, gleich in den eigenen Namensraum da alle Klassen mit Q beginnen, importiert. Dafür muss aber python-qt3 installiert sein, sonst kann das Modul nicht gefunden werden.

```
class SoyaWindow(QWidget):
    def __init__(self, *args):
        QWidget.__init__(self, *args)
        self.setCaption("Soya Application")
```

SoyaWindow wird das Fenster, welches erlaubt die Anwendung zu steuern.

```
        self.timer=QTimer()
        self.connect(self.timer, SIGNAL('timeout()'), self.update_soya)
        self.timer.start(30, 0)
```

Der Timer ist für die Hauptschleife von Soya3D verantwortlich. Durch ein sogenanntes Signal wird sie alle 30ms aufgerufen. Signale werden mit `connect` mit Slots verbunden.

```
        self.tsdisp = QTextEdit(self)
        self.tsdisp.setMinimumSize(300, 300)
        self.tsdisp.setTextFormat(Qt.PlainText)
        self.tscount = QLabel("", self)
        self.tscount.setFont(QFont("Sans", 24))
        self.forward = QPushButton("Move &Forward", self)
        self.backward= QPushButton("Move &Backward", self)
        self.quit = QPushButton("&Quit", self)
```

Hier werden die grafischen Elemente erstellt. Das ist ein Textfeld, aber auch Buttons um den Kopf nach vor und zurück zu bewegen. Zudem gibt es auch ein Knopf um das Programm zu beenden.

```
        self.layout = QGridLayout(self, 3, 2, 5, 10)
        self.layout.addMultiCellWidget(self.tsdisp, 0, 3, 0, 0)
        self.layout.addWidget(self.tscount, 0, 1)
        self.layout.addWidget(self.forward, 1, 1)
        self.layout.addWidget(self.backward, 2, 1)
        self.layout.addWidget(self.quit, 3, 1)
```

Die zuvor erstellen Objekte werden hier einem Layout hinzugefügt. Die Positionsangaben dienen zur Bekanntgabe wo genau im Layout sie platziert werden sollen. Die pixelgenaue und größenabhängige Platzierung erfolgt dann automatisch von Qt zur Laufzeit wenn die Fenstergröße feststeht.

```
        self.connect(self.forward, SIGNAL("clicked()"),
            self.move_forward)
        self.connect(self.backward, SIGNAL("clicked()"),
            self.move_backward)
        self.connect(self.quit, SIGNAL("clicked()"),
            self.close)
```

Hier werden noch die letzten Signale mit den Slots verbunden. Als Slots kann in Python übrigens jede normale Methode verwendet werden.

Bei dem Signal `clicked()`, wenn die Buttons gedrückt werden, wird jeweils eine Methode des Objektes ausgeführt.

```
    def update_soya(self):
        soya.MAIN_LOOP.update()
```

Hier ist das besagte `update()` der Hauptschleife.

```
    def move_forward(self):
        head.y += 1
        stamp = time.ctime()
        self.tsdisp.append("forward: " + stamp)
        self.tscount.setText(str(self.tsdisp.lines()))
```

Das nach vorne Bewegen funktioniert intuitiv. Es kann einfach `head.y` modifiziert werden. Zusätzlich wird noch ein `stamp` erzeugt, welche dann in dem Textfenster eingetragen wird.

```
    def move_backward(self):
        head.y -= 1
        stamp = time.ctime()
        self.tsdisp.append("backward: " + stamp)
        self.tscount.setText(str(self.tsdisp.lines()))
```

Das Rückwärtsfahren funktioniert genau gleich.

```
if __name__ == "__main__":
    soya.init()
    soya.path.append(os.path.join(os.path.dirname(sys.argv[0]), "data"))

    scene = soya.World()
    head = soya.Body(scene, soya.Model.get("caterpillar_head"))

    light = soya.Light(scene)
    light.set_xyz(2.0, 5.0, 0.0)

    camera = soya.Camera(scene)
    camera.set_xyz(0.0, 15.0, 15.0)
    camera.look_at(head)
```

Hier wird soya initialisiert. Diesen Code kennen wir schon von den ersten Beispielen.

```
    # Creates a widget group, containing the camera and a label showing the FPS.
    soya.set_root_widget(soya.widget.Group())
    soya.root_widget.add(camera)
    soya.root_widget.add(soya.widget.FPSLabel())
```

Auch die Funktionsweise des FPSLabel wurde bereits bei den Grundlagen erklärt.

```
    soya.MainLoop(scene)
```

Neu ist allerdings, dass die Hauptschleife nicht gestartet, sondern nur erzeugt wird.

```
    # Create a Qt window
    # When the button is clicked, the head moves up.

    app=QApplication(sys.argv)
    app.connect(app, SIGNAL("lastWindowClosed()"), app, SLOT("quit()"))
```

Dieser Teil hat wieder mit Qt zu tun. Hier wird die Applikation erstellt, die für jedes Programm benötigt wird.

```
w=SoyaWindow()
app.setMainWidget(w)
w.show()
app.exec_loop()
```

Letztendlich erzeugen wir noch das Objekt, der zuvor geschriebenen Klasse, setzen es als Hauptfenster, und starten die Hauptschleife von Qt.

12.2 Tkinter

Tkinter ist ein GUI-Toolkit, das standardmäßig bei Python dabei ist und damit auch recht verbreitet ist. Es ist möglich, Soya3D über Tkinter anzusprechen. Das Problem hierbei ist jedoch wie zuvor, dass sowohl Tkinter als auch Soya3D jeweils über ihre eigenen Hauptschleifen verfügen. Es ist allerdings möglich nur die von Tkinter zu verwenden und in regelmäßigen Abständen ein Update an Soya3D zu schicken. Tkinter bietet dazu die Methode `after` an, die nach einer vorgegebenen Zeit eine Funktion oder Methode aufruft. Es folgt ein Beispielprogramm:

Listing 12.2: [soya-with-tk-1.py]Tkinter

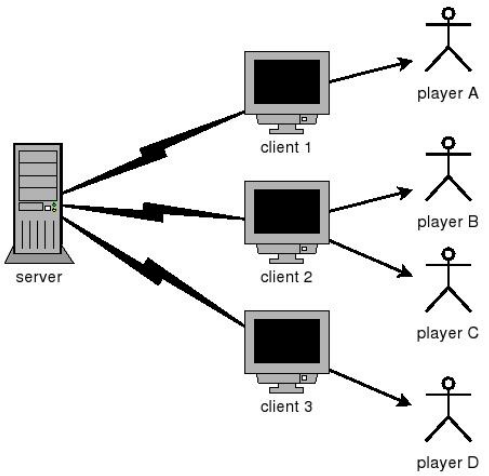
```
# .... soya specifischer Code

# Die Hauptschleife wird erzeugt, aber nicht gestartet!
soya.MainLoop(scene)

class Window(Tkinter.Tk):
    def __init__(self):
        Tkinter.Tk.__init__(self)
        self.button = Tkinter.Button(self,
            text = "Move upward", command = self.button_click)
        self.button.pack()
        self.after(30, self.update_soya)
    def button_click(self):
        head.y += 1
    def update_soya(self):
        self.after(30, self.update_soya)
        soya.MAIN_LOOP.update()
window = Window()
Tkinter.mainloop()
```

`self.after` sollte vor `soya.MAIN_LOOP.update()` aufgerufen werden. Es ist nämlich häufig der Fall, dass `soya.MAIN_LOOP.update()` Zeit verbraucht, die jedoch nicht mit-eingerechnet werden soll.

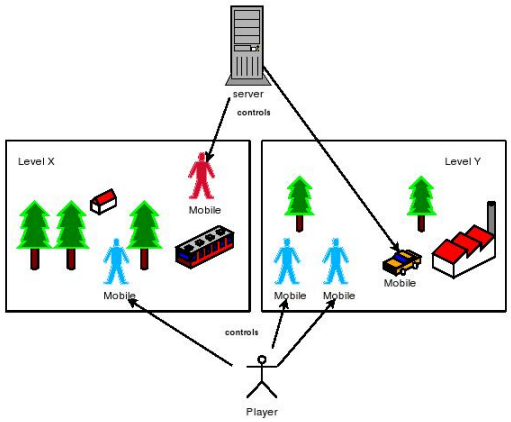
13 Soya im Netzwerk

Dieser Abschnitt ist vom Tutorial getrennt zu betrachten. Tofu, welches im folgenden beschrieben wird, führt eine zusätzliche Abstraktion ein, man verwendet Soya nicht mehr direkt so wie in den anderen Kapiteln. Es handelt sich um eine Kurzeinführung und groben Überblick um entscheiden zu können, ob Tofu für die eigene Applikation verwendet werden sollte oder nicht.

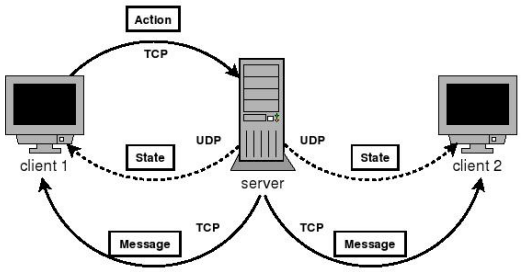
Für viele Applikationen und Spiele ist es gefordert verschiedene Kameras in einer Welt durch ein Netzwerk zu verbinden. Dies wird in Soya mit Tofu erleichtert. Ausgetauscht werden Informationen transparent über verschiedene Nachrichtentypen, wie Aktionen, Statusänderungen und andere Nachrichten.



Wie an diesem Modell ersichtbar wird, gibt es einen zentralen Server, welcher Nachrichten, eingezeichnet als Blitze, zwischen verschiedenen Computer austauscht. An einem Client können aber durchaus mehrere Spieler spielen.



Ein Spieler kann dabei verschiedene sog. Mobile steuern, dies kann auch in verschiedenen Level geschehen. Selbstverständlich können Mobile auch servergesteuert sein.



Noch ein paar technische Details zu den Austauschmechanismen. Die verschiedenen Nachrichtentypen bestehen aus Strings, die ausgetauscht werden. Es werden sowohl TCP als auch UDP sockets verwendet. Während TCP Verbindungen über eine längere Zeit aufrecht erhalten bleiben, sind UDP Nachrichten ohne Status und können verloren gehen. Statusnachrichten werden damit übertragen und nicht angekommene UDP Pakete müssen mit Interpolation aufgewogen werden.

13.1 Setup

```
import soya, soya.tofu as tofu, cerealizer
soya.path.append(os.path.join(os.path.dirname(sys.argv[0]), "data"))
tofu.SAVED_GAME_DIR = config.lookup("/sw/app/gamedir")
```

Die Initialisierung geschieht ähnlich wie bei Soya. Es muß nur zusätzlich das Modul soya.tofu geladen werden und ein *Serialisierer* wie cerealizer. Der Datenpfad ist wie gehabt, neu dazu gekommen ist der Pfad, welcher angibt, wo die Spiele gesichert werden.

```
tofu.set_side("server")
tofu.set_side("client").
```

In Tofu dient der selbe Code für alle Möglichkeiten im Netzwerk. Der erste Modus ist für Einzelspieler, der zweite ist der Server und schließlich der Client.

```
  def __init__(self, filename, password, character_name = "tux"):
    tofu.PlayerID.__init__(self, filename, password)
    self.character_name = character_name
  def dumps(self):
    return tofu.PlayerID.dumps() + len(self.character_name) + "\n" + self.character_name
  @classmethod
  def loads(Class, s):
    self = tofu.PlayerID.loads(Class, s)
    length = int(s.readline())
    self.character_name = s.read(length)
    return self
tofu.LOAD_PLAYER_ID = PlayerID.loads
```

Dieser Code erzeugt eine PlayerID. PlayerID ist eine eindeutige Kennungsmarke für einen bestimmten Spieler im Netzwerk. Der Konstruktor wird aufgerufen mit einem Dateinamen und einem Passwort, danach wird der Charaktername gesetzt.

Die `dumps` Methode retourniert die PlayerID gesichert als ein String.

Die Methode loads hingegen retourniert die PlayerID von einer Datei geladen. Für den Lesevorgang wird das Objekt s verwendet.

```
  def __init__(self, player_id):
    tofu.Player.__init__(self, player_id)
    mobile = Mobile()
    mobile.level = tofu.Level.get("first_level")
    mobile.set_xyz(100.0, 0.0, 100.0)
    self.add_mobile(mobile)
tofu.CREATE_PLAYER = Player
cerealizer.register(Player, soya.cerealizer4soya.SavedInAPathHandler(Player))
```

Für eine PlayerID können dann mehrere Player, das sind wirkliche menschliche Spieler, erstellt werden. Der Konstruktor benötigt die `player_id` welche an Player übergeben wird. Danach wird ein mobiles Objekt kreiert, welches in `first_level` an den Koordinaten (100,0,100) platziert wird.

13.2 Mobile

Mit `add_mobile` können zu einem Spieler bewegte Gegenstände hinzugefügt werden. Wichtig ist dabei, dass die Klasse weiterhin serialisierbar bleibt. Es gibt auch das Gegenstück `remove_mobile` um dem Spieler die Kontrolle eines Mobile wieder zu entziehen.

Achtung `Level.add_mobile` darf nicht aufgerufen werden, das passiert automatisch, wenn Player das Level betritt.

```
  def __init__(self):
    tofu.Level.__init__(self)
cerealizer.register(Level, soya.cerealizer4soya.SavedInAPathHandler(Level))
```

Diese Methoden für das Hinzufügen oder Löschen von Mobile kann auch überschrieben werden. Level sind immer aktiv, wenn zumindest ein von einem Spieler kontrolliertes Mobile die Welt betritt.

Tofu wurde noch lange nicht erschöpfend behandelt, man könnte ein eigenes Tutorial darüber verfassen und auch ein minimales Tofu-Programm kommt schon zu einer gewissen Länge. Dafür steht aber, dass die komplette Netzwerkkommunikation abgenommen wird.

14 Vergleich mit anderen Engines

14.1 Panda3D

Panda3D[8] ist wie Soya3D eine freie Spiele-Engine. Sie wurde von den Disney Studios entwickelt, um damit das 3D-Online Spiel Toontown zu realisieren und nimmt den Spieleentwicklern einen Großteil der immer wiederkehrenden Aufgaben ab, wie das Laden von Spielfiguren und grundlegende Bewegungssteuerung. Geschrieben ist Panda3D größtenteils in C++, aufgrund der größeren Performance. Allerdings wird diese Funktionalität üblicherweise über Python angesprochen. Zwar ist es auch möglich gleich in C++ zu programmieren, es wird jedoch davon abgeraten, da es einiges an Funktionalität gibt, die nur in Python existiert.

Es gibt ein Panda3D-Format, Egg genannt, das für die Modelle verwendet wird. Es gibt Export-Plugins für Maya, Softimage XSI, 3D Studio Max, sowie Blender. Für Blender gibt es drei verschiedene Plugins mit unterschiedlichen Beschränkungen. Am weitesten verbreitet ist Chicken. Zum Ausprobieren der Engine werden jedoch auch einige Modelle und Welten mitgeliefert.

Panda3D ist Soya3D recht ähnlich, unterscheidet sich jedoch in einigen Punkten. So

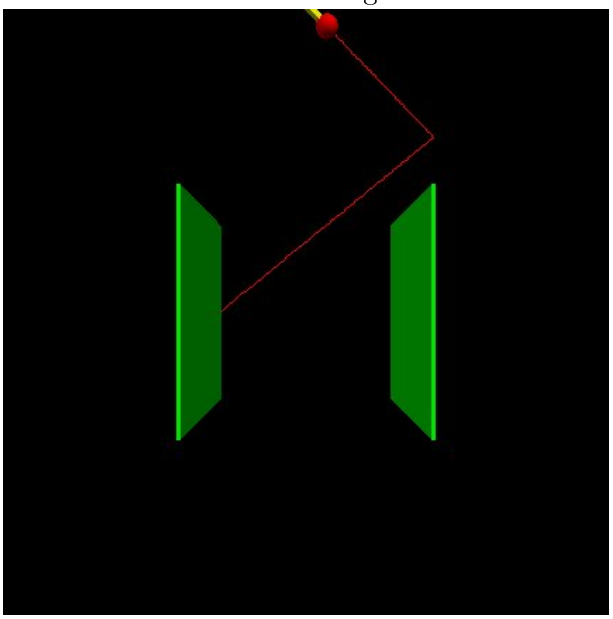
hat Panda3D zum Beispiel nur eine rudimentäre Physik-Engine implementiert. Soya3D hat außerdem die Möglichkeit Welten in Welten zu modellieren, so dass es zum Beispiel einfach möglich ist den Mond um die Erde und die Erde um die Sonne kreisen zu lassen. Probleme dieser Art und auch physikalische Simulationen sind mit Panda3D komplizierter.

Auf der anderen Seite hat Panda3D eine bessere Integeration von Ereignissen. Während Soya3d hier mit SDL-Konstanten arbeitet, hat Panda3D seine eigene Implementierung, die einfacher zu handhaben ist.

Die in Panda3D eingebaute Sound Engine FMOD steht nicht unter einer freien Lizenz. Wer dies wünscht muss sich eine Alternative suchen, zum Beispiel die Soundfunktion von pygame. Die Soundfunktion von Soya3D integriert sich dagegen sehr gut und bietet Stereo Sound ohne auf die Nachteile einer proprietären Lizenz.

14.2 VPython

VPython[27] steht für Visual und Python. Visual ist ein 3D Graphik Modul für Python, entwickelt von David Scherer. Damit ist es möglich einfache 3D Objekte, wie z.B. Kugeln, Kegel oder Zylinder zu zeichnen und zu animieren. Es besitzt ein eigenes Modul um Maus oder Keyboardereignisse abzufangen. Allerdings gibt es im Gegensatz zu Soya3D oder Panda3D keine Möglichkeit 3D-Models zu verwenden, die man beispielsweise in Blender gezeichnet hat. Hierzu müsste man die Models selber laden und verwalten, indem man Faces zeichnet, was jedoch in den meisten Fällen den Aufwand nicht rechtfertigt und man besser auf eine andere Engine verwenden sollte.

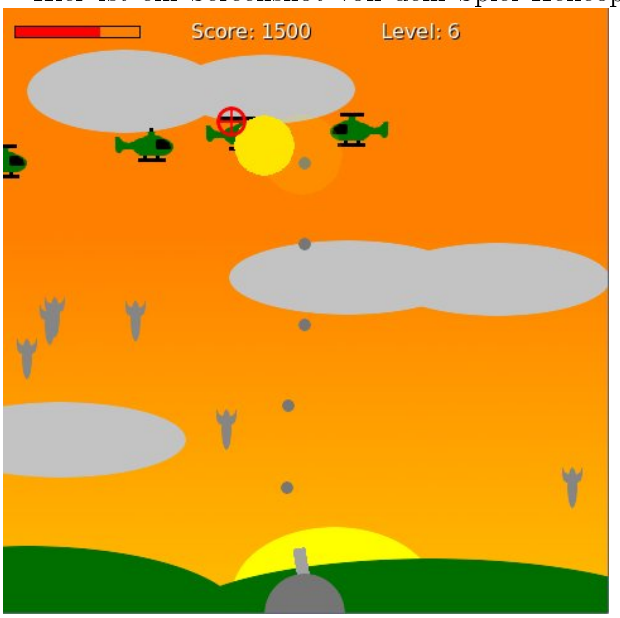


Dennoch gibt es für Visual Anwendungsgebiete und zwar überall dort wo man keine Models verwenden will, wie zum Beispiel für Spiele wie Tetris oder um ein physikalisches Experiment zu illustrieren.

14.3 Pygame

Pygame[18] verwendet als Grundlage SDL. Es hat gute Unterstützung zur Behandlung von Ereignissen und Sound. Es ist plattformunabhängig hat allerdings keine 3D Unterstützung. Geeignet ist es daher hauptsächlich für 2D-Spiele. Dennoch findet es bei 3D-Programmierung Anwendung, da es sehr gute Unterstützung für Interaktionen und Sound hat. Oder man verwendet Pygame für die 2D-Elemente, wie zum Beispiel die Menüs.

Hier ist ein Screenshot von dem Spiel Helicopters, das mit pygame realisiert wurde:



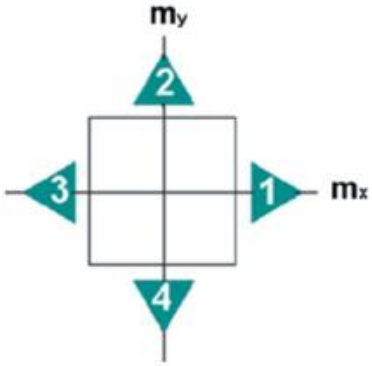
15 Programme die Soya3d verwenden

Soya3D, ursprünglich für Spiele konzipiert, hat mittlerweile auch seine Bedeutung für wissenschaftliche Software. Dieses Kapitel stellt Programme vor, welche in Python mit Soya3D geschrieben sind oder es für Teilbereiche verwenden.

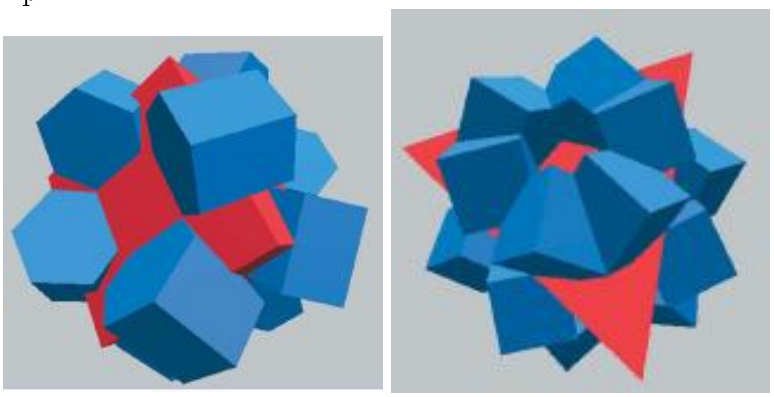
Diese Sektion kann wertvolle Tipps geben für welche Einsatzgebiete Soya3D geeignet ist und beschreibt auch Probleme, wenn welche aufgetreten sind.

15.1 GenOVa

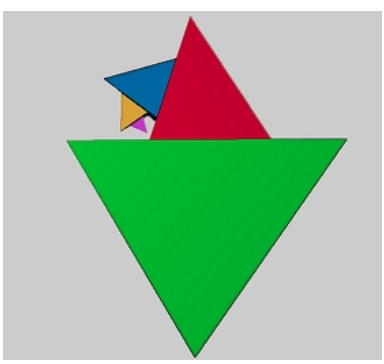
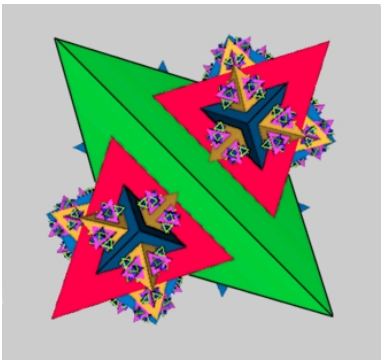
Diese wissenschaftliche Software beschäftigt sich mit Materialen und deren Eigenschaften. Dabei wurde auch eine Visualisierung in der das Besagte mit vielen Würfel dargestellt wird mit Soya3D realisiert. [3]



Diese 2D Grafik macht nur schlecht klar, wie die Eigenschaften in einem tatsächlichen Körper sind.



Hingegen sieht man in der 3D Visualisierung, wie die genaue Orientierung und Geometrie der Eigenschaften aussieht.

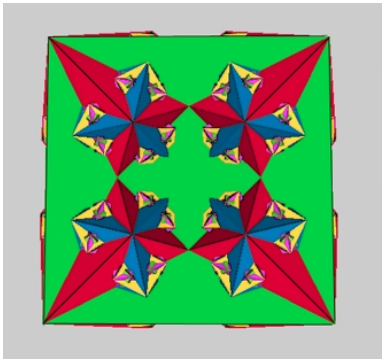
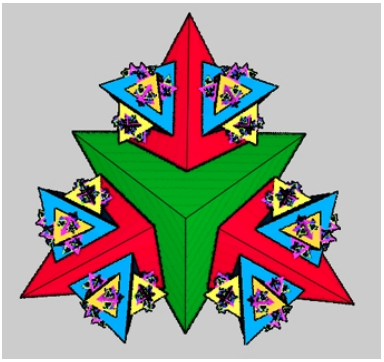
Bei diesen nachträglich eingefärbten Bildern sieht man, wie genau welche Körper für die 3D Repräsentation verwendet wurden, und zwar nur Würfel und Tetraeder. [4]

Cyril Cayron, der Autor des Programmes sagt über Soya3D:

> For 3D representations, Soya3D is so easy and complete that what I wrote is very short and basic. For 3D representations, I just linked my crystallographic algorithms (that are the core of GenOVa), used functions that were already written in Soya3D, and created menus.

auf Deutsch übersetzt lautet es etwa:

> Für 3D Repräsentationen ist Soya3D so einfach und vollständig, dass das Programm welches ich geschrieben habe sehr kurz und einfach sein konnte. Ich habe einfach meine kristallografischen Algorithmen (welche der Kern von GenOVa sind) mit Funktionen von Soya verknüpft und die Menüs dazu erstellt.

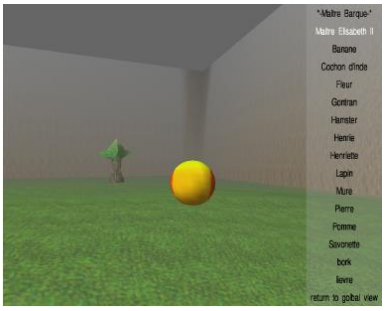



15.2 Multi-Agenten Systeme

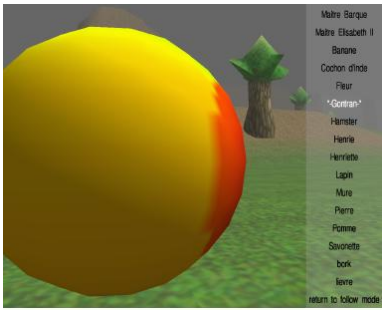
Eine andere wissenschaftliche Arbeit beschäftigt sich mit der Visualisierung von Multiagentensystemen. Dabei kann ein Level kreiert werden und die Agenten, als Bälle dargestellt, beobachtet werden. [20]

Dabei wurden sogar 4 verschiedene Ansichten realisiert. Die globale Übersicht ermöglicht es den Level zu überblicken:

Es gibt allerdings auch eine Ansicht von einer 3rd Person Perspektive, die sogenannte Drittpersonansicht. Der Vorteil davon ist dass man auch noch das Level hinter dem Agent eingeschränkt sieht und früher Gegenstände auf der Seite wahrnimmt:



Neben einer Kontrollübersicht wurde auch die 1st Person Perspektive realisiert:



15.3 Packet Garden

Diese Anwendung von Julian Oliver ist die wohl interessanteste, welche auf Soya3D basiert. Es handelt sich hier zwar auch um eine reine Visualisierung ohne Interaktionen, die Daten werden aber von einem lokalen Paketfilter gewonnen. Es wird dargestellt wie man das Internet verwendet.

15.3.1 Installation

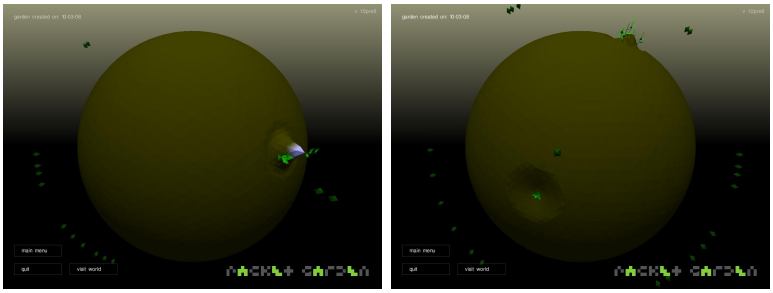
Die Software ist Opensource und wie Soya3D unter GPL lizenziert. Für Debian Testing und Unstable liegen direkt Pakete vor, die nach der Installation der Abhängigkeiten einfach mit ”dpkg -i” installiert werden können.

Das Programm wird wie üblich direkt als Python-Programm ausgeliefert und sowohl unter Linux, MacOSX als auch Windows getestet.

15.3.2 Funktionalität

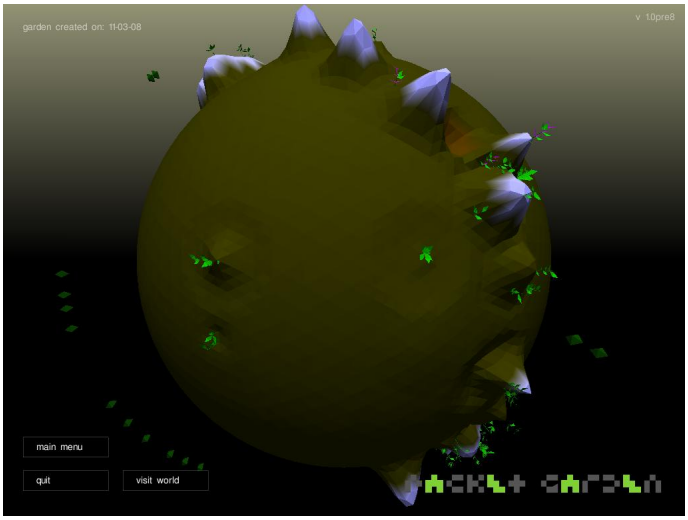
Alle besuchten Server werden zusammen mit deren geografischen Lokalisierung abgespeichert. Uploads werden zu Hügel und Downloads zu Tälern.

Pflanzen hingegen stellen dar welche Protokolle verwendet werden. Greift man auf eine Webseite zu, wächst eine HTTP Pflanze. Diese sind natürlich auf der Spitze der Berge oder inmitten eines Tales.

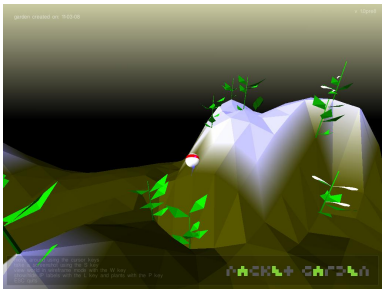
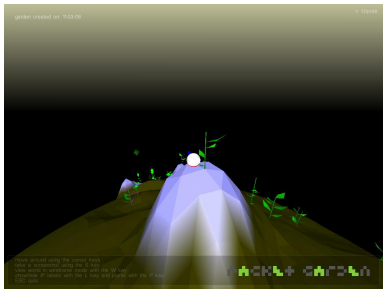


Diese Informationen bleiben natürlich vollkommen lokal und werden in einer Datei im Home Directory gespeichert. Hier wird aber auch eine Geschichte früherer Gärten gespeichert, die später angeschaut werden können.

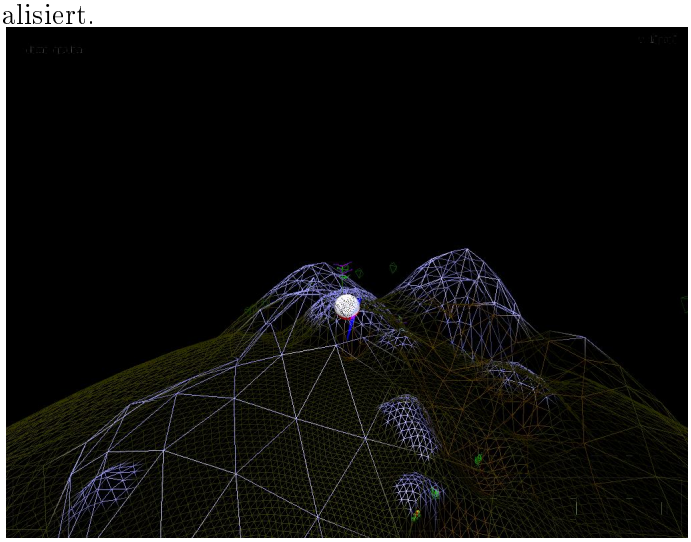
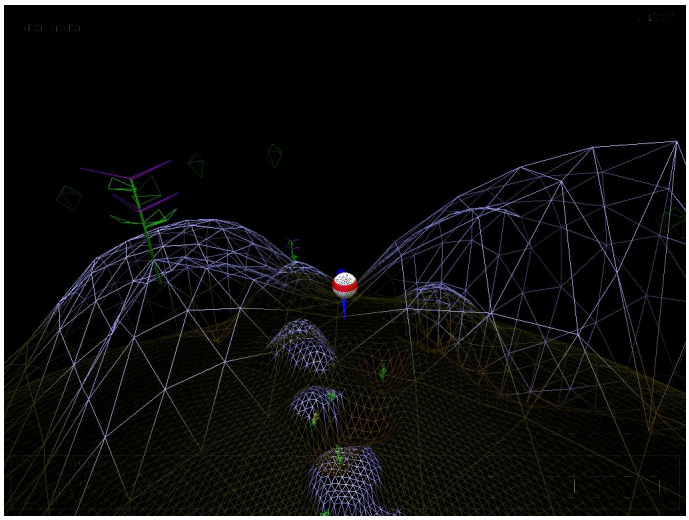
Damit kann man sein eigenes Verhalten im Internet wie in einem Tagebuch anschauen. Besonders intressant ist Peer-To-Peer Traffic da hier mit Hunderten von Peers auf der ganzen Welt Daten ausgetauscht werden. Die Pflanzen dabei sind dann violett.



Diese Welt kann auch noch begangen werden. Hier kann man mit den Cursortasten eine Kugel steuern.

Um auch durch die Welt durch blicken zu können wurde auch ein Wireframe Modus realisiert.

Die IP Adressen und Länder lassen sich auch einblenden.

15.3.3 Technisches

Packetgarden verwendet für die Menüs pudding, siehe Kapitel 9. Dabei wurde aber die default style class überschrieben, um einen Fehler zu umgehen. Dazu musste OpenGL verwendet werden.

Die Welt selbst wird mit Terrains (siehe Kapitel 8.4) realisiert. Die großen Unterschiede im Datenvolumen bringen aber Probleme mit sich. Während Downloads von Filmen im GB Bereich sind, sind Textseiten nur wenige kB. Der erste Ansatz des Authors war eine Sigmoid Funktion, welche oftmals bei neuralen Netzwerken [23] verwendet wird, die auch wesentlich bessere Ergebnisse als ein linearer Zusammenhang erreichte. In der

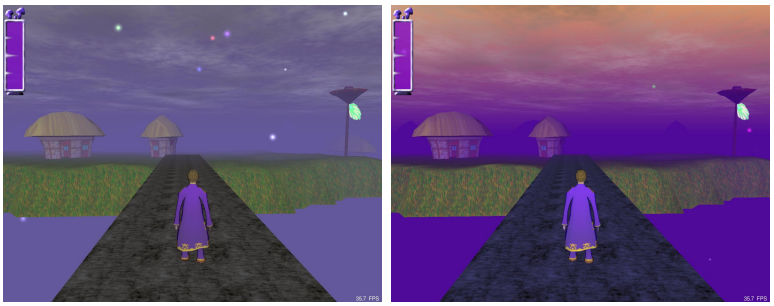
aktuellen Version werden die Größen allerdings nur linear sortiert. Damit ist es möglich zu erkennen, welches Volumen größer ist als ein anderes, aber nicht um wieviel.

Das ganze Programm ohne Filterung sind nur 1516 Zeilen Python Code. Viele davon sind aber nur die Reimplementierung von pudding.

15.4 Balazar

Dieses ist das erste Spiel welches in Soya3D realisiert wurde, hat aber mittlerweilen beeindruckende Grafik und verwendet die meisten Features von Soya3D. Der Protagonist Balazar begleitet auch durch viele Tutorials von Soya3D.

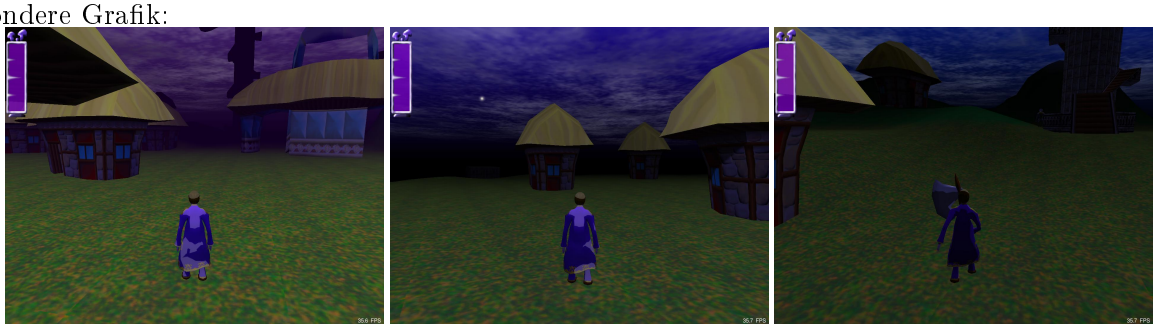
15.4.1 Grafik



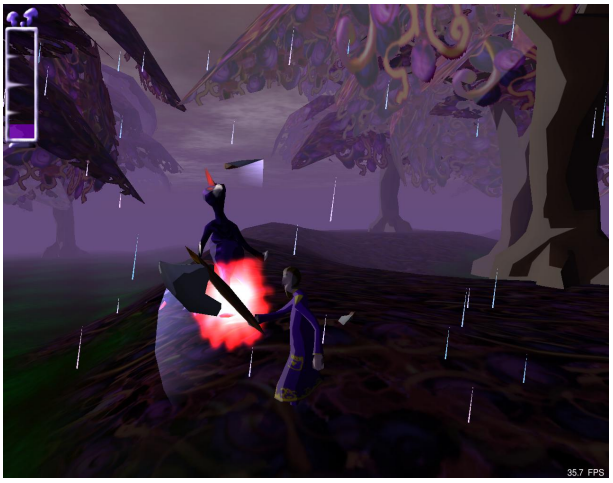
Die Veränderungen von Farben stechen sofort in das Auge und sind sehr stark für die gut gelungene Stimmung verantwortlich.

Das Spiel verwendet Tofu, siehe Kapitel 13. Die Levelübergänge sind mit langen Brücken realisiert:

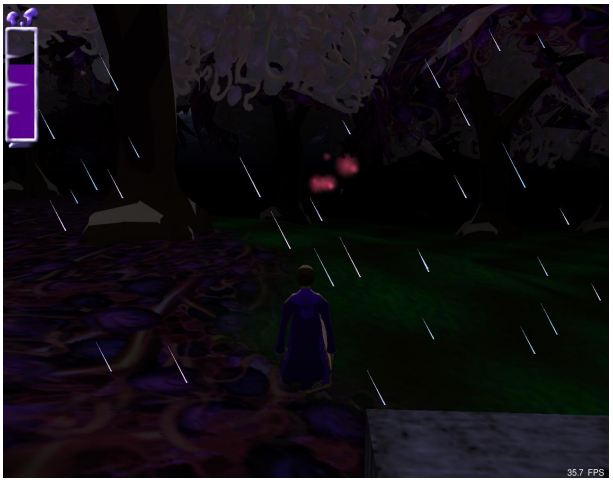
Die folgenden Bilder zeigen verschiedene Wettersituationen und verdeutlichen die besondere Grafik:



In dem Rollenspiel muss man sich gegen futuristische Gegner verteidigen. Dabei wurde auf Kampfanimationen besonderer Wert gelegt. Es ist auch möglich verschiedene Schwerter und Äxte zu haben.



In der Nacht im Wald kann man kaum etwas erkennen.



15.4.2 Technik

Die Technik ist in diesem Spiel wesentlich komplexer und schwieriger als in den zuvor vorgestellten Projekten. Zu einem wurden auch KI Gegner realisiert und es ist auch ein

serverbasiertes Multiplayer Spiel. Insgesamt ist das Spiel in 27 Dateien mit je ca. 500 Zeilen Code.

Die Lebensanzeige ist noch direkt in OpenGL realisiert, pudding gab es bei der Erstellung des Spieles noch nicht. Da diese Leiste aber ständig im Vordergrund ist, genügen nur einige opengl Befehle.

Die Menüs sind mit soya.widget gemacht. Sie sehen aber sehr einfach aus obwohl sie einiges an Aufwand benötigen. Diese Vorgehensweise wird deshalb nicht mehr empfohlen.

Die Levels und das Wetter werden selbst generiert und kommen ohne Verwendung von Soya3D Funktionen aus. Das Spiel verwendet *Cel Shading* um comichaft zu wirken. Die wenigen Abstufungen kann man hier gut am Balazar und den Bäumen erkennen.

Dieses Spiel demonstriert, dass Soya3D auch für größere Projekte und Programme verwendbar ist. Zudem kann es als Referenz für Tofu gesehen werden. Leider ist wie bereits erwähnt die Komplexität höher, und es benötigt einige Einarbeitungszeit um den Quellcode zu verstehen.

15.5 Balazar Brothers

Dieses Spiel hat mit dem vorigen nur den Hauptdarsteller gemein. Es handelt sich hier nicht um ein Rollenspiel, sondern ein Nachdenkspiel ohne direkte Steuerung des Charakters. Die zwei Brüder werden mit der linken und rechten Shifttaste zum springen veranlasst. Dabei können sie aber nur auf eine Säule gleicher Höhe oder niedriger springen. Um andere Hindernisse zu überwinden müssen die zwei Brüder so geschickt zusammenarbeiten, dass zumindest einer den jeweiligen Ausgang erreicht.

15.5.1 Rundgang durch das Spiel

Die Balazar Figuren können allerdings nicht zurückspringen, so hat man in dem unten gezeigten Szenario bereits verloren.

Die Aufgabe ist es 7 Schlüssel für die nächste Türe einzusammeln.

Die Welten sind sehr unterschiedlich gestaltet, so sieht der Fluss der Fontänen wieder anders aus.

Hier sind man die Charaktere wie sie gerade springen. Auch hier wird *Cel Shading* verwendet. Bei diesem Screenshot sieht man wie der Charakter zur Verdeutlichung schwarz umrahmt ist.

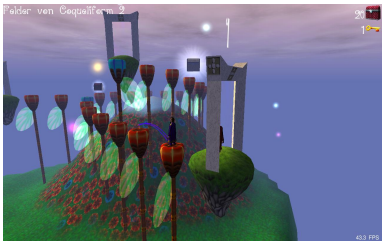
Die Animationen sind hier wesentlich ausgefeilter. So kann ein Balazar auf den anderen springen und auf ihm balancieren.

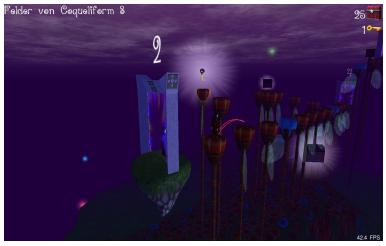
Am eindrucksvollsten ist die unten gezeigte Spezialbewegung. Opfert man einen Balazar so springt er in den Tod. Steht aber gerade der andere auf seinen Schultern, so kann

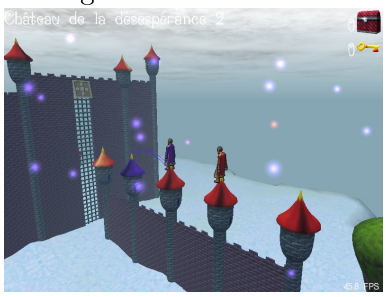
dieser zum Ausgang segeln:

Auch hier gibt es verschiedene Wettersituationen.

Zum Schluss noch ein paar Bilder, welche die restlichen Welten zeigen:







15.5.2 Technisches

Auch technisch ist dieses Spiel Balazar sehr ähnlich gehalten, hat allerdings eine Generalüberholung erhalten. Es wird zwar auch Tofu verwendet, neu ist allerdings dass auch Terrains verwendet werden.

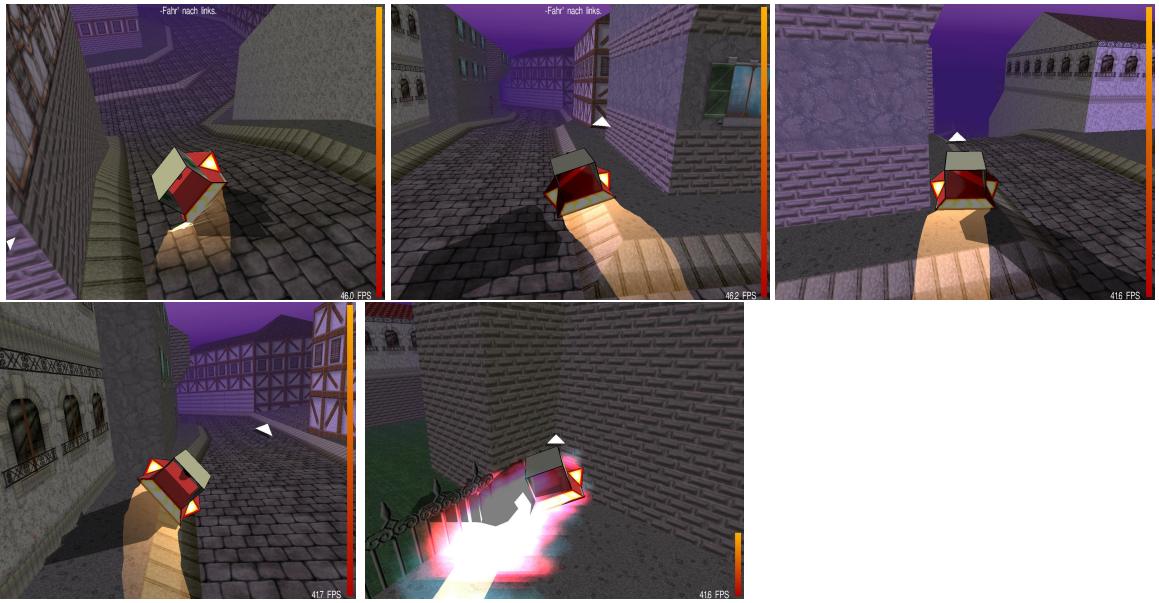
Die Komplexität ist allerdings niedriger, so sind es jetzt nur noch 19 Dateien mit durchschnittlich 400 Zeilen, die auch einfacher zu lesen sind. Selbstverständlich werden hier aber mehr Funktionen von Soya3D genutzt.

15.6 Slune

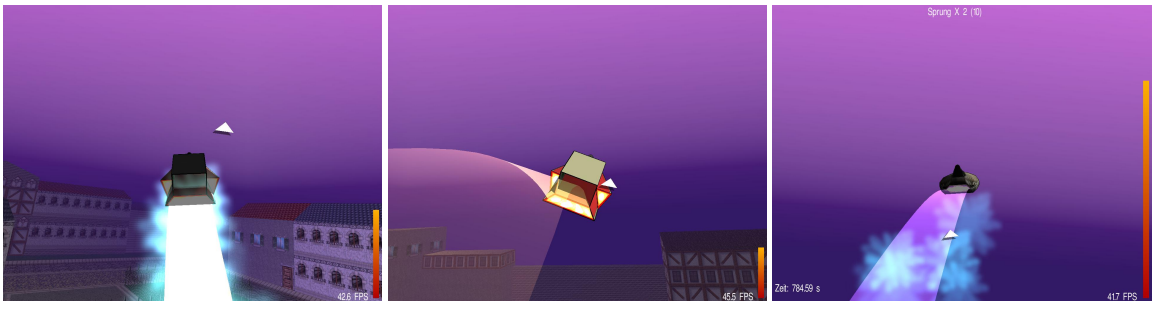
Dieses Spiel ist nicht nur wie die zwei vorigen vom Soya3D Entwickler Jean-Baptiste Lamy geschrieben worden, sondern auch von Bertrand Lamy. Das geschah im Rahmen

von Nekeme Prod. In dem Spiel wird auch eine politische Nachricht transportiert.

Es ist möglich mehrere Fahrzeuge zu steuern. Durch die hier verwendete Physik Engine ODE (siehe 8.1) wurde eine beeindruckende Fahrzeugsimulation geschaffen, in der auch Unfälle möglich sind.

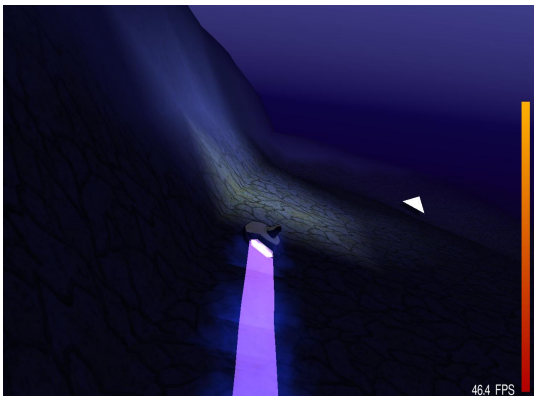


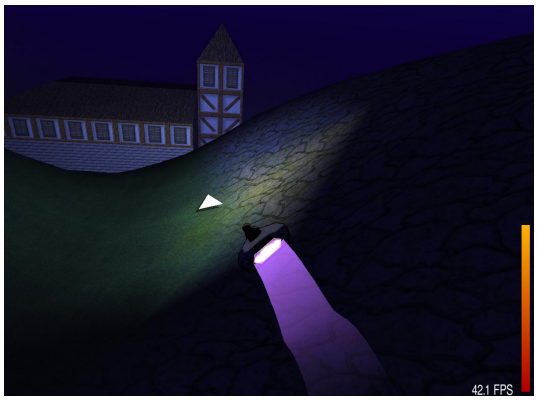
So sind aber auch sehr hohe Sprünge möglich. Der Streifen hinten am Fahrzeug welcher ein Geschwindigkeitsgefühl gibt, ist transparent gehalten:



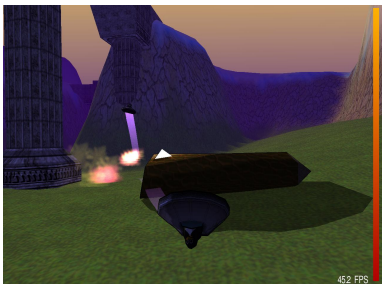
Auch hier gibt es wieder Gegner, die durch KI gesteuert werden.

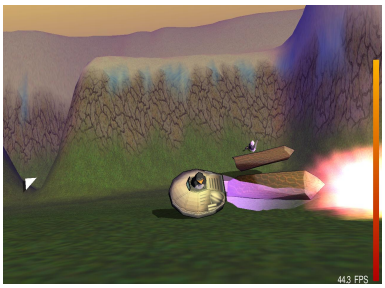
Bei Nachtfahrten wird das Licht des Fahrzeuges eingeschaltet:

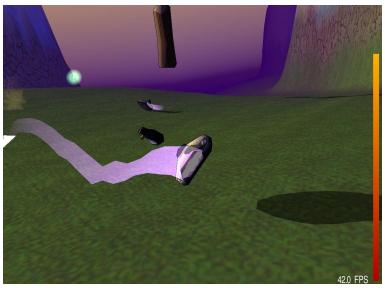




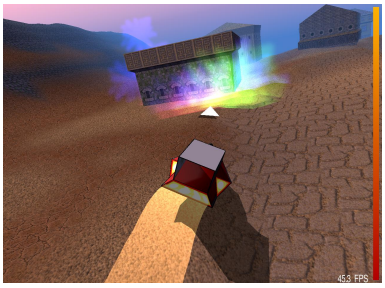
Die Physik Engine ist insbesondere in einem Level beeindruckend, in dem der Gegner versucht, Baumstämme auf einen zu werfen:







Im Spiel gibt es auch eine Waffe, die sehr farbenfroh und transparent ist.



Die letzen 3 Screenshots zeigen die Reise nach Afrika auf einem Flugzeug und die Darstellung von dem Land. Die großen Landschaften die dafür notwendig waren, wären ohne Terrains nicht möglich gewesen:

Dieses Spiel gehört zu den aufwendigsten in Soya3D geschriebenen mit 12155 Zeilen in 29 Dateien. Dies ist insbesondere darauf zurückzuführen dass das Spiel sehr dialoglastig und mit vielen Levels ausgestattet ist.

15.7 Collège 3D

Dieses Spiel wurde von Pascal Peter und seinen Schülern geschrieben. Dabei kann man Schüler und Lehrer in einer nachgebauten Schule bewegen. Hier wird auch pygame verwendet. Das Menü ist mit Pudding realisiert.

15.8 Sage

Sage ist eine Open Source Software für Mathematik. Anstatt eine neue Sprache zu erfinden wird Python verwendet. Die 3D Visualisierung dabei kann auch mit Soya3D erzeugt werden, standardmäßig wird allerdings gnuplot verwendet.

16 Danksagung

Wir möchten uns bei folgenden Personen bedanken:

- Cayron Cyril (GenOVa Bilder)
- Souvarine (Sound, Raypicking)
- und allen anderen die uns bei diesem Tutorial geholfen haben

17 Glossar

Python muss für Soya3D in der Version 2.2 oder höher installiert sein. Es wurde auch mit Python 2.4 getestet.

Python ist eine dynamische, objektorientierte und interpretierte Programmiersprache.

Pyrex ist eine eigene Sprache, um für Python Erweiterungsmodule zu schreiben. In ihr ist es möglich Python und C Datentypen nach belieben zu mischen und das in eine C Erweiterung für Python zu kompilieren.

Der Hauptvorteil liegt darin, nicht den relativ umständlichen C-Code schreiben zu müssen, der für Python C-Module notwendig ist. Der Prozess wird darauf simplifiziert nur die .h Dateien mit zusätzlichem Code und Annotationen umschreiben zu müssen.

Das Problem von C-Modulen in Python liegt in den Typsystemen begraben. Während C Typen während der Compilierung überprüft, wird dieser Schritt in Python erst zur Laufzeit bei der Verwendung der Variablen gemacht.

OpenGL ist die am weitesten verwendete, am besten unterstützte und am besten dokumentierte 2D/3D Graphik API. Wie sonst auch üblich kann auch bei Soya eine beliebige Implementation der API verwendet werden, meistens die der Grafikkartenhersteller oder des Betriebsystems.

Ein besonderen Wert wird auf Stabilität der API gelegt, so war es nie notwendig dass Soya diese Teile umschreiben musste. Es sind auch direkte Aufrufe von OpenGL Befehlen von Programmen welche Soya3D verwenden möglich. Auch hier profitiert man von den besonderen Kompatibilität. OpenGL 3.0 wird eine komplett neue API einführen, aber auch die alten Methoden weiter mitschleppen.

Während, wie im Tutorial ausführlich besprochen, fast alles komplett ohne OpenGL Befehle realisiert werden kann, benötigt man sie um Soya3D zu erweitern und nicht vorgesehene 3D Ausgaben zu realisieren.

```
soya.init ("App with opengl")
```

Soya3D führt einen kompletten Import von Opengl Methoden unter soya.opengl durch. Mit import as können diese dann direkt mit `opengl` angesprochen werden. Damit stehen einem alle OpenGL Befehle innerhalb einer Soya Applikation in Python zur Verfügung.

OpenAL ist eine plattformunabhängige 3D Audio API in besonderer Rücksicht auf Bedürfnisse von Spiele entwickelt. Mit ihr ist es möglich Audio quellen in einem 3D Raum zu modellieren.

Die API ähnelt der von OpenGL. Sie erlaubt sowohl lineare als auch exponentielle Distanzen zu modellieren und hat Unterstützung für den Dopplereffekt.

GLEW ist eine Bibliothek mit der portabel festgestellt werden kann, welche *OpenGL* extensions auf der aktuellen Plattform zur Verfügung stehen. Damit ist es des weiteren möglich die genaue OpenGL Version zu bestimmen.

Es wird zudem das Programm glewinfo mitgeliefert, mit dessen Hilfe man alle fehlenden und vorhandenen Extensions auf seinem System anschauen kann.

SDL ist eine platformunabhängige Library welche low level access zu Audio, Tastatur, Maus und 3D Hardware mit OpenGL anbietet. Sie ist sehr einfach zu verwenden und leistet gute Arbeit beim Abstrahieren von Hardware und ist deshalb sehr bei Spielen beliebt.

Dies wird in Soya3D benötigt, da *OpenGL* nur 3D Ausgabe ohne etwas von dem verwendeten Device oder Fenster etwas zu wissen unterstützt.

Neben der Ereignisbehandlung, Audioausgabe werden auch Threads und Zeitgeber von SDL verwaltet.

Cal3D ist eine Bibliothek, welche sich mit 3D Charakter Animation beschäftigt. Sie ist vom gleichen Autor wie Soya. Dabei hat jedes Modell ein Skelett, welches sehr viele Freiheiten gibt. SDL Container werden benutzt, um Daten zu speichern. Erzeugen der Charaktere ist aufwendig und z.b. mit 3D Studio MAX möglich. Dabei muss neben dem Skelett auch der Schwerpunkt und wie das Objekt modifiziert werden kann, festgelegt werden.

libFreeType2 ist eine font engine, welche hoch qualitative Glyphenbilder erzeugen kann ohne dass irgendeine Arbeit mit Dateien in denen die Schriften beschrieben werden manuell verwendet werden müssen. Von diesen Glyphenbilder können zudem monochrome bitmaps oder anti-aliased pixmaps erzeugt werden. Dabei werden 256 Grautöne verwendet.

Es handelt sich um ein modulares System welches neue Schriftformate nachladen kann. Verwendet wird es neben Soya3D auch für Textlayout, Paginierung und für Font Inspektions- und Konversionstools.

FreeFont erzeugt anspruchsvolle und hoch qualitative Schriften unter GNU GPL für viele character sets.

Python Imaging Library unterstützt viele Bildformate und erlaubt vielfältige Nachbearbeitung, wie zurecht schneiden, rotieren oder aber auch abspeichern, da Python von sich aus keine Bilder verarbeiten kann.

In Soya3D wird diese Bibliothek verwendet, um Texturen zu erschaffen.

Blender ermöglicht es Körper zu modellieren und anschließend für Soya3D zu exportieren.

Serialisierer ist eine objektorientierte Technik ein Objekt welches gerade im Speicher ist so zu repräsentieren, dass es zu einem späteren Zeitpunkt wiederhergestellt werden kann. Während die einfachen Datentypen meistens einfach zu Strings umgewandelt werden können muss der Serialisierer mehr Arbeit bei Referenzen investieren um sie korrekt wiederherstellen zu können.

Siehe auch *Pickle* und *cerealizer*.

Pickle ist ein Modul für Soya für die Serialisierung und De-serialisierung von Python Objekte. Dabei werden beliebige Python Objekte in ein Stream verwandelt der dann gespeichert oder über Netzwerk verschickt werden kann.

Siehe auch *cerealizer*.

cerealizer unterstützt die gleiche Funktionalität wie *Pickle* ist allerdings in Python geschrieben und auch auf Sicherheit bedacht. Trotz allem gibt kaum Unterschiede in der Performance.

ODE wird im Zusammenhang mit Kollisionsentdeckung im Kapitel 8.1 erklärt.

yet-in(complete) ist eine englische Abkürzung für „noch immer unfertig" welches der Name des englischen Tutorials ist.

Ogre ist wie Soya3D eine objektorientierte 3d-Engine. Sie geht allerdings in der Funktionalität weit über Soya3D hinaus, ist allerdings wesentlich komplexer und umfangreicher. Während Ogre in C++ geschrieben ist, könnten Applikationen auch in Python durch Ogre-Bindings geschrieben werden.

Ogre zielt aber auf größere und professionellere Applikationen ab. Es werden wesentlich mehr Dateiformate und 3D Grafik APIs unterstützt.

pygame ist eine Menge von Python Modulen, die helfen Spiele zu entwickeln. Auch sie bauen auf die *SDL* library auf.

Cel Shading wird verwendet, wenn es nicht erwünscht ist, exakte und realistische Schatten zu produzieren. Stattdessen wird eine Ästhetik im Stil von Zeichentrickfiguren realisiert. Die Technik ist durchaus auch aufwendig, kann aber steril wirken. Richtig und gezielt eingesetzt hingegen kann sie einen anspruchsvollen künstlerischen Aspekt gerecht werden.

Gimp ist ein Bildbearbeitungsprogramm, welches zum Erstellen und Ändern von Texturen sehr gut geeignet ist.

Konvex ist ein Körper oder eine Fläche dann, wenn alle inneren Winkel kleiner als 180° sind. Diese Eigenschaft ist für Schnittpunkte, wie bei Raypicking siehe Kapitel 11 benötigt, vorausgesetzt.

Normalvektor einer Fläche ist das Kreuzprodukt von zwei Kantenvektoren.

Siehe Mathematik in Soya3D im Kapitel 4.

Tkinter ist ein Wrapper für das Tk GUI-Toolkit. Es war das erste GUI-Toolkit für Python und gehört auch zum Lieferumfang. Zwar gibt es viele andere Toolkits, aber dadurch das Tkinter fast überall verfügbar, ist hat es eine recht große Bedeutung.

CoordSyst ist die Basisklasse für alle 3D Objekte in Soya3d.

Listings

Literaturverzeichnis

[1] Ahikhmin, Michael: *Synthesizing Natural Textures.* Symposium on Interactive 3D Graphics, 2001.

[2] Baker, Hearn: *Computer Graphics with OpenGL.* Pearson Prentice Hall, neue Auflage, 2004.

[3] Cayron, Cyril: *GenOVa: a computer program to generate orientational variants*, 2007. `http://journals.iucr.org/j/issues/2007/06/00/cg5071/cg5071.pdf` (10.3.2008).

[4] Cayron, Cyril: *Multiple twinning in cubic crystals: geometric/algebraic study and its application for the identification of the $\Sigma 3^n$ grain boundaries.* Acta Crystallographica Section A, 63(1):11–29, Jan 2007.

[5] Chen, Bo und Harry H. Cheng: *Interpretive OpenGL for computer graphics.* Computers & Graphics, 29(3):331–339, June 2005.

[6] Drmota, M., B. Gittenberger, G. Karigl und A. Panholzer: *Mathematik für Informatiker.* Heldermann Verlag, 2007.

[7] Foley, James D., Andries van Dam, Steven K. Feiner und John F. Hughes: *Computer Graphics: Principles and Practice in C.* Addison-Wesley Professional, zweite Auflage, August 1995.

[8] Frommel, Oliver: *3-D Welten mit Python und Panda3D.* Linux User, November 2006.

[9] G. Baron, P. Kirschenhofer: *Einführung in die Mathematik für Informatiker.* Springer, 1989.

[10] Havlicek, Hans: *Lineare Algebra für Technische Mathematiker.* Heldermann Verlag, 2006.

[11] Hess, Roland: *the Essential Blender.* Blender Foundation, erste Auflage, 2007.

[12] Kaiser, Peter und Johannes Ernesti: *Python Das umfassende Handbuch.* Galileo Computing, aktuell zur Version 2.5 Auflage, 2007.

[13] Kaufmann, Thomas: *Einführung in Python.* Linux User, Mai 2003.

[14] Kraft, Johann, Eugen Hruby, Heinrich Bürger, Hubert Unfried und Stefan Götz: *Mathematische Formelsammlung.* öbvhpt, 2003.

[15] Kylander, Karin und Olof S. Kylander: *GIMP - Das offizielle Benutzerhandbuch.* mitp, 1999.

[16] Lamy, Jean-Baptiste: *The (yet-in) complete guide to Soya 3D.* 2007.

[17] Lamy, Jean-Baptiste: *Soya3D Homepage*, 2008. `http://home.gna.org/oomadness/en/about_me___/index.html` (24.2.2008).

[18] McGugan, Will: *Beginning Game Development with Python and Pygame.* Apress, 2007.

[19] Panne, Joseph Laszlo Michiel van de und Eugene Fiume: *Interactive Control For Physically-Based Animation.* SIGGRAPH 2000 Conference Proceedings, 2000.

[20] Pierre-Yves, David: *Projet tuteure Cration dune interface multi-agents*, 2007. `http://marmoute.free.fr/iut/projtuto-rapport.pdf` (10.3.2008).

[21] Pilgrim, Mark: *Dive Into Python.* Apress, zweite Auflage, 2004.

[22] Rempt, Boudewijn: *GUI Programming with Python: QT Edition.* Command Prompt, 2001.

[23] Saul, Lawrence K., Tommi Jaakkola und Michael I. Jordan: *Mean field theory for sigmoid belief networks.* Journal of Artificial Intelligence Research, 4:61–76, 1996.

[24] Schwarzer, Stefan: *Einstieg in Python - Programmieren lernen in fünf Schritten.* Linux User, September 2006.

[25] Schwarzer, Stefan: *Gut verschürt - Funktionen, Module und Pakete in Python.* Linux User, Oktober 2006.

[26] Schwarzer, Stefan: *Vererbungslehre - Fehlerbehandlung und objektorientiertes Programmieren in Python.* Linux User, November 2006.

[27] Schwarzer, Stefan: *Durchstarten - Hilfen für das Programmieren in Python.* Linux User, Februar 2007.

[28] Schwarzer, Stefan: *Einfache Architektur - Objektorientiertes Programmieren in Python.* Linux User, Januar 2007.

[29] Snyder, John M.: *Interval analysis for computer graphics.* Computer Graphics, 26(2):121–130, 1992.

[30] Unbekannt: *Blender Tutorial*, 2008. `http://www.crowle.de/blender_tutorials/blender_tutorials.html` (24.2.2008).

[31] Wikipedia: *Physik-Engine - Wikipedia, Die freie Enzyklopädie*, 2007. [Stand 2. Januar 2008].

[32] Wikipedia: *Kreuzprodukt - Wikipedia, Die freie Enzyklopädie*, 2008. [Stand 24. Februar 2008].

[33] Zilm, Thorsten: *LaTeX Das Einsteigerseminar*. moderne industrie Buch, erste Auflage, 2003.