MNU

Der mathematische und naturwissenschaftliche Unterricht

05
Jahrgang 64
Juli 2011 · € 7,85

Naturwissenschaft im Kontext

Während Modellieren und Rechnen Prozesse sind, stellt die interaktive Simulation ein Produkt dar. Gelungene Simulationen können eine Belohnung sein und sind durch ihre leichte Bedienbarkeit auch für Laien zugänglich.
Des Weiteren bedienen die interaktiven Simulationen einen für heutige junge Menschen wichtigen Aspekt ihres Lebens: leichte Verfügbarkeit und Verteilbarkeit. Die interaktiven Simulationen können innerhalb weniger Sekunden über eine Email oder einen Blog, Twitter oder Link in einem sozialen Netzwerk weiterverbreitet werden und zu Diskussionen und Entdeckungen einladen.

Literatur

ABDEL-AZIZ, N. (2009). *The Greatest Fenced Area along a Barn.* The Wolfram Demonstrations Project: *http://demonstrations.wolfram.com/TheGreatestFencedAreaAlongABarn/* (22.07.2010).

COCHRAN, L. (2009). *The Disk Method.* The Wolfram Demonstrations Project: *http://demonstrations.wolfram.com/TheDiskMethod/* (22.07.2010)

JONG, M. D. (2010). *Motion of the Moon: Phases.* The Wolfram Demonstrations Project: *http://demonstrations.wolfram.com/MotionOfTheMoonPhases/* (22.07.2010).

KUEHN, N. & SCHERBAUM, F. (2010). *Seismicity of Germany.* The Wolfram Demonstrations Project: *http://demonstrations.wolfram.com/SeismicityOfGermany/* (22.07.2010).

ROSEMOND, P. (2009). *A Basic Property of Integrals.* The Wolfram Demonstrations Project: *http://demonstrations.wolfram.com/ABasicPropertyOfIntegrals/* (22.07.2010).

Stellungnahme der GDM sowie des Deutschen Vereins der MNU zur »Empfehlung der Kultusministerkonferenz zur Stärkung der mathematisch-naturwissenschaftlichen Bildung«. *http://www.kmk.org/fileadmin/veroeffentlichungen_beschluesse/2009/2009_05_07-Empf-MINT.pdf*, (22.07.2010).

Q -Magazin für Arbeit und Lernen-Digitales Klassenzimmer (2010). Die Welt entdecken und Erforschen. Sonderdruck S. 3.

WEIGAND, H.-G. (2010). Computereinsatz in den MINT-Fächern – Ja, natürlich aber wie viel darf's denn sein? *MNU, 63,* 259.

WÖLFER, A. (2010). *Berechnung von statistischen Kenngrößen.* The Wolfram Demonstrations Project: *http://demonstrations.wolfram.com/BerechnungVonStatistischenKenngroeszen/* (22.07.2010).

Prof. Dr. HERBERT HENNING, henning@ovgu.de, Otto-von-Guericke-Universität Magdeburg, Institut für Algebra und Geometrie (Didaktik der Mathematik

ANDREJ WÖLFER, andrej.w@me.com, Wilhelm-Niemann-Str. 19, 39112 Magdeburg, ist Lehramtsstudent an der Otto-von-Guericke-Universität Magdeburg.

Modellieren und Darstellen von 3D-Objekten

Welche Ziele des Informatikunterrichts lassen sich mit PovRay und Visual Python erreichen?

REINHARD OLDENBURG – JÜRGEN POLOCZEK

3D-Computergrafik ist ein attraktives Thema, mit dem man verschiedene wichtige Ziele im Unterricht der Sekundarstufe I verfolgen kann. Um in Abhängigkeit von diesen Zielen das richtige Werkzeug auswählen zu können, werden in diesem Beitrag zwei vielversprechende Kandidaten, PovRay und Visual Python, vorgestellt und die damit gegebenen didaktischen Möglichkeiten eingeordnet. Neben der Auswahlhilfe verfolgt die Arbeit – gewissermaßen auf der Meta-Ebene – noch ein zweites Ziel: Es soll gezeigt werden, welche Bildungsziele Informatikunterricht besonders gut verfolgen kann und welche Stellung der Informatik im Kanon der naturwissenschaftlich-mathematischen Fächern zukommt.

1 Einleitung

In vielen Bundesländern gibt es keinen Lehrplan für das Fach Informatik in der Sekundarstufe I, häufig wird es nur im Bereich des Wahlunterrichts angeboten, ist also nicht fest in der Stundentafel verankert. Ein solcher Status als freiwilliges Fach stellt Informatik in einen gewissen Wettbewerb mit anderen Fächern, um die knappe zu verteilende Unterrichtszeit bewerben sich mehrere Fächer. In der Oberstufe hat Informatik inzwischen einen höheren Stellenwert, aber es tritt das Problem geringer Schülerzahlen auf. Dies hat u. a. auch einen Grund darin, dass es an »Nachwuchs« aus der Sekundarstufe I mangelt. Um sich in der Konkurrenzsituation behaupten zu können, ist qualifizierter, attraktiver Informatikunterricht notwendig. In einer Befragung von Schülerinnen und Schülern, durchgeführt von OLDENBURG & RABEL (2009), wurde eine Vielzahl von Projekten genannt, die diese gerne durchführen würden. Zu den am häufigsten genannten Themen gehörte die 3D-Grafik. Dieses Thema erhält zusätzliches Interesse durch die aktuellen Trends zu 3D-Kino und 3D-Fernsehen.

2 Das Programm PovRay

Den Wünschen der Schüler nach ästhetisch schönen 3D-Grafiken kann man mit dem Programm PovRay entsprechen, und es bietet auch eine Reihe von Aspekten, die für Informatikunterricht an sich wichtig sind. Besonders lassen sich viele Bezüge zu den Bildungsstandards der Gesellschaft für Informatik, 2008, herstellen. Nicht vergessen sollte man auch die Tatsache, dass nicht nur an den Gymnasien Informatikunterricht in der Sekundarstufe I angeboten wird.

2.1 Was ist PovRay?

Der Name PovRay ist eine Abkürzung für **P**ersistance **o**f **V**ision **Ray**tracer. Persistance of Vision bedeutet sinngemäß »Die Beharrlichkeit/Trägheit des Sehens/Nachbildes (auf der Netzhaut)« oder »Die Fortdauer des Eindrucks« oder »Nachleuchten eines Bildes«. Diese Mehrdeutigkeit der Interpretation ist offenbar von den Autoren erwünscht. Raytracing enthält die Vokabeln »ray« (Strahl) und »to trace« (verfolgen), bedeutet also so viel wie »Stahlenverfolger«. Raytracing stellt einen auf der Methode der Strahlenaussendung basierenden Algorithmus zur Berechnung der Sichtbarkeit von dreidimensionalen Objekten von einem bestimmten Punkt aus dar. Die Grundidee ist die der Umkehrung des Sehens in der Strahlenoptik. Physikalisch gesehen, reflektiert jeder Punkt eines Objekts das von einer oder mehreren Lichtquellen stammende Licht in alle Richtungen. Sieht man den Gegenstand, so gelangt von diesen Strahlen ein Bündel ins Auge. Zur Berechnung der Sichtbarkeit ist dieses Modell nicht gut geeignet, weil es zu rechenintensiv wäre, »alle Strahlen«, die ein Objekt aussendet, zu berechnen. Ein Raytracer kehrt deswegen die Situation (den Strahlengang) um, es wird für jedes Pixel die Richtung, aus der das Licht ins Auge fällt, berechnet und anschließend die Intensität und Farbe des Pixels bestimmt. Hierbei machen Verdeckungen und auch Reflexionen die Berechnung aufwändig. Insgesamt wird aber der Rechenaufwand verringert.

Diese knappe Darstellung zeigt, wie viel Physik in einem solchen Programm steckt. Wenn die Schüler mit den Optionen, die es bietet, spielen, kommen sie beispielsweise auf Oberflächeneigenschaften wie den Unterschied von diffuser und gerichteter Reflexion. Wenn transparente Objekte erzeugt werden, kommen die Gesetze der Optik besonders deutlich zum Ausdruck, etwa wenn man einen Glasquader vor einem Bild modelliert. Die spektrale Zerlegung wird aber nicht modelliert: Ein virtuelles Prisma zeigt keine Farben! Ein schöner Anlass über Zwecke und Grenzen von Modellen nachzudenken.

PovRay ist ein 3D-Computergrafikprogramm, welches es ermöglicht, dreidimensional wirkende Szenen am Computer »fotorealistisch« zu erstellen. Die Bilder lassen sich in sehr hoher Auflösung und sehr guter Farbbrillanz »rendern«. Hierzu dient eine eigene Skriptsprache. Darzustellende Gegenstände werden aus Einzelteilen zusammengesetzt, die durch mathematische Funktionen beschrieben und über Schlüsselwörter implementiert werden. Beispielsweise gibt es zum Erzeugen einer Kugel die Anweisung *sphere* {Parameter}. Neben den Objekten muss man eine in eine bestimmte Richtung weisende Kamera und mindestens eine Lichtquelle, die die Gegenstände beleuchtet, definieren.

Die nun von PovRay zu berechnende Abbildung dieser räumlichen Szene in ein Bild ist ein recht komplexer Vorgang, denn die in der Szene vorhandenen Körper werden nicht nur direkt durch die vorhandenen Lichtquellen beleuchtet, es gibt durchaus auch indirekte Beleuchtung, die bspw. durch Reflexionen an anderen Körpern entsteht. Auch Schattenräume sind zu berücksichtigen, ebenso die mögliche Transparenz einiger Gegenstände. All diese Erscheinungen werden beim Einsatz von PovRay einbezogen und erklären, warum das Rendern auf hoher Qualitätsstufe relativ lange dauern kann.

2.2 Ein erstes Beispiel

Nach dem Start von PovRay wird zunächst ein neues, leeres Dokument erzeugt, dann stellen wir die Größe in Pixel des später zu erzeugendes Bildes (Pop-down-Menü, links oben) auf bspw. 800 x 600 ein, es ist die Einstellung »AA« (»Anti-Aliasing«) möglich, was eine höhere Bildqualität, aber auch erhöhten Rechenaufwand zur Folge hat.

Nun fügen wir als erste Zeile der neuen Datei *#include »colors.inc«* ein. Hiermit erreichen wir, dass nunmehr Farben mit vordefinierten Namen benutzt werden können (ansonsten müssten Farben über ihren RGB-Anteil definiert werden). Es ist leicht einsehbar, dass eine Farbgebung *{color Gold}* anstelle von *{color rgb <0.8, 0.498039, 0.196078>}* deutlich vorteilhafter zu handhaben ist. Welche Farbbezeichnungen das im Einzelnen sind, kann man durch Öffnen der Include-Datei, die im Dokumentenordner abgespeichert wird, nachlesen.

Nun setzen wir die Hintergrundfarbe fest, z. B. durch *background {White}*. Wie oben beschrieben, müssen nun eine Kamera und mindestens eine Lichtquelle platziert werden. Dies geschieht mittels der Anweisungen:
camera {location <10,10,–10> look_at <0,0,0> angle 50}
light_source {<–20,10,–20> color White}

Die in spitzen Klammern angegebenen Zahlen haben die Bedeutung von Koordinaten im dreidimensionalen Raum, es wird das in Abbildung 1 gezeichnete (linkshändische) Koordinatensystem benutzt, die übrigen Parameter (*location* – Standpunkt des Betrachters bzw. der Kamera, *look_at* – Blickpunkt, auf den die Kamera gerichtet ist, *angle* – Öffnungswinkel der Kamera) erklären sich nahezu selbst.

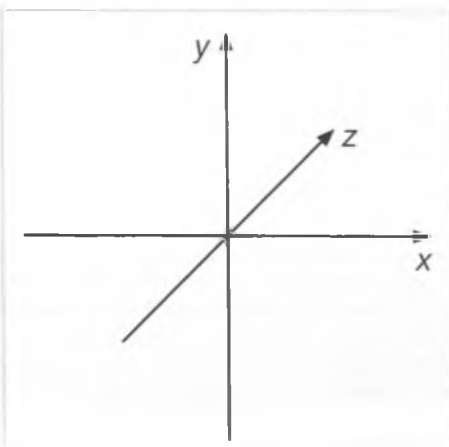

Abb. 1. Linkshändisches Koordinatensystem

Nun fügen wir der Szene eine Ebene hinzu, dies geschieht mittels *plane {<0, 1, 0> 0.5 pigment {color Red}}*. <0, 1, 0> ist der Normalenvektor, er steht orthogonal auf der Ebene, diese wiederum liegt also zur *xz*-Ebene im Koordinatensystem parallel. Die Zahl 0.5 gibt den Abstand zum Ursprung an. Mit *pigment {…}* wird die Farbe und fakultativ die Transparenz festgelegt. Für Letzteres ergänzt man hinter dem Parameter *color* den optionalen Parameter *transmit* mit Werten zwischen 0 und 1.

Zum Abschluss ergänzen wir die Szene um eine Kugel *sphere {<1, 2, –2> 3 pigment {color Aquamarine transmit 0.7}}*, wobei wir auch einen Parameter namens *texture* einsetzen können, um vorgefertige Farbmuster (Texturen) verwenden zu können. Für Texturen gibt es, wie auch für die Farben, eine Include-Datei *(»textures.inc«)*, die ebenfalls eingebunden werden sollte. <1, 2, –2> bezeichnet die Koordinaten des Kugelmittelpunktes, die Zahl 3 gibt den Radius an.

Eine Textur wird durch die folgenden drei Bestandteile vollständig beschrieben:

1. Farbeigenschaft:, inklusive Durchsichtigkeit: *pigment {…}*
2. Oberflächenstruktur (Rauheit): *normal {…}*
3. Helligkeits-, Glanz- und Reflexionseigenschaften: *finish {…}*

Beispiel:

```
sphere {<0,2,–2> 3 texture {Lightning2}
                   normal {ripples 0.1}
                   finish {diffuse 0.9 phong 0.5}}
```

erzeugt eine leicht lichtdurchlässige, lilafarbene Kugel mit Schlieren auf der Oberfläche und Glanzlichtern.

Für einen ersten Einstieg in die Arbeit mit PovRay mag das nun genügen. Man findet bspw. Bei LOHMÜLLER (2010) findet man einen sehr gut gestalteten Kurs mit einigen Beispielen zum Nachmachen, Abändern, Ausbauen usw.

Mögliche Beispiele für den Unterricht findet man in großer Anzahl durch eine einfache Recherche, es liegt also eine Fülle an Erfahrungen mit dem schulischen Einsatz in verschiedenen Bereichen, nicht nur dem Informatikunterricht, vor.

2.3 PovRay im Informatikunterricht der Sek 1

Sicherlich liegt der Schwerpunkt in einem Informatikkurs der Sekundarstufe 1 im Bereich der Modellierung. Schüler gestalten die Szenen durch systematischen Aufbau, ausgehend von einigen wenigen Grundelementen. Sie nutzen den Freiraum, der sich durch die Variation der Parameter ergibt und vertiefen hierbei ihr Verständnis. Gleichzeitig erhalten sie einen Einblick in die Arbeit mit einer einfachen Skriptsprache und deren Syntax. Die »Constructiv Solid Geometry« als eine Grundlage von PovRay erlaubt das Arbeiten mit Körpern nach den Regeln der Mengenlehre. Es gibt die Möglichkeit zum Bilden von Durchschnitt, Vereinigung und Differenz mehrerer Körper, um daraus neue Formen zu schaffen. Das räumliche Vorstellungsvermögen der Schüler wird hierdurch gefördert, wie auch durch die Verwendung eines (linkshändischen) dreidimensionalen Koordinatensystems, in dem sie ihre Elemente sinnvoll platzieren müssen.
Sie sind gezwungen, syntaktisch korrekt zu arbeiten, eventuell auftretende Fehler werden sofort angezeigt, das Bild kann in einem solchen Fall nicht gerendert werden. Auch Modellierungsfehler werden deutlich, da PovRay unmittelbar die visuelle Rückmeldung gibt, ob das erzeugte Bild den eigenen Vorstellungen entspricht oder nicht. Die erzeugten, ästhetisch anspruchsvollen Produkte dienen der Motivation, sich an Neuem zu versuchen und so seine Kenntnisse zu erweitern.
Betrachten die Schüler professionell mit PovRay gestaltete Bilder, so erhebt sich die Frage nach der »Echtheit« von Bildern. Ist das, was wir in Zeitschriften sehen nun wirklich ein Foto oder »synthetisch«? Auch dies muss mit einem Kurs diskutiert werden.

In dem weiteren Verlauf einer Unterrichtseinheit zu PovRay werden sicherlich Variablen, Entscheidungen (*if*- und auch *switch*-Anweisungen) sowie Wiederholungen (*while*-Schleifen) thematisiert werden, womit man schon die Grundelemente einer Programmiersprache hat. Mittels der *declare*-Anweisung definiert man seine eigenen Objekte (nicht im Sinne der Informatik, aber »nahe dran«) und kann diese mehrfach in einer Szene verwenden, *Include*-Dateien erlauben es, bestimmte Programmteile »auszulagern« und damit modular zu arbeiten – ein wichtiges Konzept der Informatik.

2.4 Ein zweites Beispiel

Einige der oben erwähnten Möglichkeiten sind im zweiten Beispiel zusammengefasst, in welchem deren Verwendung demonstriert wird.
Zunächst nehmen wir grundlegende Einstellungen vor und führen die notwendigen *Include*-Dateien auf, um sinnvoll mit Farben und Texturen arbeiten zu können.

```
global_settings {assumed_gamma 1.0}[1]
#default{finish{ambient 0.1 diffuse 0.9}}
#include »colors.inc«
#include »textures.inc«
```

Um die Verwendung einer Mehrfachauswahl zu demonstrieren, werden fünf unterschiedliche Kamerapositionen definiert, wobei mithilfe des Selektors Kamera ausgewählt wird. Mittels *case* werden die Positionen für die Werte eins bis vier festgelegt, der *else*-Teil bestimmt die Lage bei Eingabe anderer Ziffern. Hierbei werden die für die Kamera wichtigen Parameter in den Variablen *Position*, *Auf* und *Winkel* gespeichert. Prinzipiell deklariert man mit *#declare* globale, mittels *#local* lokale Variablen, z. B. in einem Makro oder einer *Include*-Datei.

```
#declare Kamera = 2;
#switch (Kamera)
#case (1)                          //Breitseite
  #declare Position = < 2,4,–4>;
  #declare Auf = < 2, 1, 3>;
  #declare Winkel = 55;
#break
#case (2)                          //seitlich, mehrere Gläser
  #declare Position = < 5.00, 5.00, –5.00>;
  #declare Auf = < 0.5,1.5,2>;
  #declare Winkel = 12;
#break
#case (3)                          //ein Glas, groß
  #declare Position = < 0,5,0>;
  #declare Auf = < 0,0,2>;
  #declare Winkel = 15;
#break
#case (4)                          //Schmalseite
  #declare Position = < –5,5,3>;
  #declare Auf = < 2,0,3>;
  #declare Winkel = 50;
#break
#else
  #declare Position = < –3, 3, –7>;
  #declare Auf = < 0, 1, 1>;
  #declare Winkel = 25;
#end

camera{location Position
    look_at Auf
    angle Winkel}
```

[1] LOHMÜLLER empfiehlt diese Einstellungen, damit die Farben nicht zu dunkel dargestellt werden.

Es wird die Position zweier Lichtquellen festgelegt, die zweite, weit entfernte, soll die Sonne darstellen.

```
light_source {<0,5,5> color White}                  //Lampe
light_source {<–1000,1000,–1000> color White}       //Sonnenlicht
```

Ein Objekt *Tisch* setzt sich aus einer Holzplatte *(box)* mit vier Füßen *(cylinder)* zusammen. An dieser Stelle wird der Tisch noch nicht gezeichnet, sondern nur beschrieben. Der Zeichenvorgang wird mittels der Anweisung *object(Tisch)* ausgeführt (s. u.).

```
#declare Tisch = object {
union {
  box {<–1,1,1> <5,1.2,5> texture {Yellow_Pine}}//oder
  DMFWood4, Sandalwood
  cylinder {<–0.8,0,1.2> <–0.8,1,1.2> 0.1 pigment {color Gray}}
  cylinder {<–0.8,0,1.2> <–0.8,1,1.2> 0.1 pigment {color Gray}
  translate <5.6,0,0>}
  cylinder {<–0.8,0,1.2> <–0.8,1,1.2> 0.1 pigment {color Gray}
  translate <0,0,3.6>}
  cylinder {<–0.8,0,1.2> <–0.8,1,1.2> 0.1 pigment {color Gray}
  translate <5.6,0,3.6>}}
```

Analog dazu wird ein Objekt *Glas* definiert, das sich aus einem ausgehöhlten Kegelstumpf (»Differenz« zweier Kegelstümpfe) und einem weiteren, innen passenden, Kegelstumpf, der Flüssigkeit, zusammensetzt. (Der »innere« Kegel, der herausgeschnitten wird, muss eine etwas größere Höhe als der äußere besitzen, sonst scheint das Glas oben nicht offen zu sein.)

```
#declare Glas = object {
union {
difference {
  cone {<0,1.2,1.5> 0.055 <0,1.5,1.5> 0.06}
  cone {<0,1.22,1.5> 0.05 <0,1.51,1.5> 0.055}
  texture {NBglass}//alternativ: material {M_NB_Glass}}
  cone {<0,1.22,1.5> 0.05 <0,1.35,1.5> 0.055 texture {Water} finish
  {reflection 0.9}}}}
```

Bis zu dieser Stelle wurde noch nichts gezeichnet, dies erfolgt nun. Zuerst verlegen wir einen grauen Teppichboden

```
plane {<0,1,0>, 0
  texture {pigment {color Gray60}
          normal {bumps 0.5 scale 0.01}
          finish {phong 0.15}}}
```

und stellen einen Tisch darauf.

```
object{Tisch}
```

Nun erzeugen wir mittels einer *while*-Schleife vier Gläser vom oben beschriebenen Typ und verschieben sie an unterschiedliche Positionen auf der Tischplatte.

```
#declare zaehler = 0;
#while (zaehler < 4)
object{Glas translate <0,0,zaehler/2>}
#if (zaehler = 1)
object{Glas translate <zaehler,0,zaehler>}
#end
#declare zaehler = zaehler + 1;
#end
```

Unser oben definiertes Objekt Glas ist unflexibel, wir haben nur eine Form, die mehrfach dupliziert und verwendet werden kann. Um mehrere Variationsmöglichkeiten nutzen zu können, verwenden wir die Struktur *#macro … #end*, welche eine Parameterübergabe erlaubt, um die Glassorte und damit auch die Farbe des Glases bestimmen zu können. Selbstverständlich ist der Einsatz weiterer Parameter möglich.

```
#macro FarbigesGlas (Farbe)
union {
difference {
  cone {<0,1.2,1.5> 0.055 <0,1.5,1.5> 0.06}
  cone {<0,1.22,1.5> 0.05 <0,1.51,1.5> 0.055}
  texture {Farbe}}}
#end

object {FarbigesGlas (Green_Glass) translate <1,0,2>}
```

Oder auch:

```
#declare Farbe = Ruby_Glass;
object {FarbigesGlas (Farbe) translate <1,0,2.5>}
```

Die Abbildung 2 zeigt unser Produkt aus der Kameraposition 4. Mit einfachen Mitteln können Animationen erstellt werden; sie bestehen aus einzelnen Bildern, die zu einem Video zusammengesetzt werden. Dies ist beispielsweise in LOHMÜLLER (2010) gut nachvollziehbar erläutert.

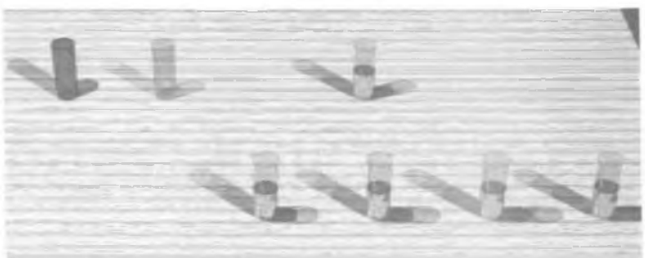

Abb. 2. Das im zweiten Beispiel erzeugte Produkt

2.5 Bezüge zu den Bildungsstandards der Informatik

Insgesamt gesehen liegen die Bezüge des Themas PovRay zu den Bildungsstandards der GI sicherlich primär beim Modellieren, aber auch andere Kompetenzen (Information und Daten, Informatiksysteme, Informatik und Gesellschaft) sind deutlich tangiert. Das Zusammenstellen einer Szene erfordert viel Sorgfalt beim Platzieren der Gegenstände, der Kamera und auch der Lichtquellen. Auch die Definition von Objekten oder Makros bedarf der Überlegung: Welche Eigenschaften sind statisch, welche dynamisch, wie setze ich meine Objekte aus anderen zusammen? Sinnvollerweise arbeitet man mit einem Modell des Koordinatensystems, um einen guten Überblick zu erhalten. Auch die Wahl der zusätzlichen Parameter und deren Werte ist hier zu beachten: Im Prinzip hat man den »Prototyp« eines Modellierungskreislaufs, der mehrfach durchlaufen wird, bis das Ergebnis der Realität bzw. der Wunschvorstellung nahe kommt. Die verwendete Skriptsprache ist syntaktisch etwas »sperrig«, allerdings gibt es viele Beispiele, an denen sich die Schüler im Unterricht orientieren können. Diese Art des Umgangs mit einem Informatiksystem entspricht der gängigen Arbeitsweise in der Informatik. Es muss nicht das Rad jedes Mal von Neuem erfunden werden.

Fotorealistische Darstellungen sind – von Profis angefertigt – kaum von »echten« Fotografien zu unterscheiden. Ein Exkurs durch verschiedene »Galerien« legt die Vermutung nahe, dass viele an anderer Stelle veröffentlichte Bilder so entstanden sein könnten. Wohin geht hier die Entwicklung?

Betrachtet man die Beispiele, die mittels einer Internetrecherche zu finden sind, so wird erkennbar, dass sich PovRay auch im Mathematikunterricht sinnvoll einsetzen lässt. Sei es zur Visualisierung von Körpern im Raum in beiden Sekundarstufen oder auch, wie es A. FILLER von der Humboldt-Universität Berlin zeigt, um die mathematischen Grundlagen von Raytracing zu beleuchten. Fachübergreifender Unterricht ist demzufolge

ebenfalls möglich. Ebenso könnte man im Rahmen eines solchen Unterrichts die Strahlenoptik vertiefend behandeln.
Unter *http://www.lehrer-online.de/povray-relativistisch.php* ist ein schönes Beispiel zu finden, in dem gezeigt wird, wie mittels PovRay kurze Videosequenzen erzeugt werden, die Auswirkungen der speziellen Relativitätstheorie visualisieren. LOHMÜLLER gibt eine Fülle von Links[2] an, die zu Anwendungsbeispielen aus verschiedenen Bereichen führen.
Auch über ein Schulbuch fand PovRay unter dem Stichwort »Vernetztes Anwenden« bereits Einzug in den Unterricht.
Bezüge zum Fach Kunst[3] sind ebenfalls vorhanden, allein zum Thema »Perspektive« bietet PovRay sehr viele Ansatzpunkte, Licht und Schatten ergänzen dies. 3D-Darstellungen von Molekülen oder auch Orbitalen bieten sich für den Chemie- und Physikunterricht an.

Eine gute theoretische Grundlage zum Thema 3D-Grafikprogrammierung, in der alle wichtigen Begriffe erklärt sind, findet man bei in MÖLLER (2008).

3 Visual Python

Visual Python, kurz VPython *(www.vpython.org)*, ist eine stark vereinfachende Schnittstelle zwischen der Sprache Python und der professionellen Grafikbibliothek OpenGL. Ziel der Entwicklung ist dabei gerade nicht, alle Möglichkeiten von OpenGL nach Python durchzureichen, sondern einen besonders einfachen Zugang zu ermöglichen.

3.1 Ein einfaches Programm

Ein einfaches Programm sieht damit so aus:

```
from visual import *
sphere(pos=(0,0,0),radius=2,color=color.white)
sphere(pos=(10,0,0),radius=1,color=color.red)
sphere(pos=(0,10,0),radius=1,color=color.green)
sphere(pos=(0,0,10),radius=1,color=color.yellow)
```

Dieses Programm erzeugt vier Kugeln mit den angegebenen Radien und Farben. Man kann diese Attribute der Objekte auch noch nachträglich mit der »Punkt-Notation« ändern, wenn man das von *sphere* erzeugte Objekt in einer Variablen speichert. Die folgende Befehlssequenz, die man in der interaktiven Python-Shell Schritt für Schritt erproben kann, erzeugt eine weitere (per *default*) weiße Kugel und verschiebt sie mehrfach um den gleichen Translationsvektor:

```
>>> b=sphere(pos=(4,1,2))
>>> b.pos=b.pos+(1,2,0)
>>> b.pos=b.pos+(1,2,0)
>>> b.pos=b.pos+(1,2,0)
```

Man kann dabei den Eindruck gewinnen, dass die Kugel sich auf einer Geraden bewegt und diese Erkenntnis lässt sich zu einer kleinen Animation ausbauen:

```
>>> import time
>>> for i in range(20): # Wiederhole das Nächste 20 Mal
        b.pos+=(–0.2,–0.2,–0.1)
        time.sleep(0.1)
```

Schließlich erweitern wir noch die Schleife um einen Befehl, der eine kleine Kugel längs der Bahn zurücklässt.

```
sphere(pos=b.pos, radius=0.2)
```

Neben Kugeln gibt es auch noch Zylinder und Boxen. Bei Zylindern gibt man den Ortsvektor des Mittelpunktes eines Deckelkreises an sowie einen Vektor, der die Achse des Zylinders angibt. Pfeile *(arrow)* werden ganz ähnlich definiert.

```
cylinder(pos=(0,2,1), axis=(5,0,0), radius=1)
```

Mit diesen Mitteln lassen sich mit wenigen Zeilen schon nette 3D-Objekte darstellen und – weil man die Szene interaktiv drehen und vergrößern kann – von allen Seiten betrachten.

Die Schüler müssen dazu mit 3D-Koordinaten umgehen können, aber diese Fähigkeit kann notfalls auch in diesem Kontext erlernt werden.

Die Voraussetzungen an die Programmiererfahrung sind denkbar gering. Man könnte sogar den Einstieg in die Programmierung mit Visual Python vornehmen:

- Objekte werden nacheinander erzeugt oder geändert: Idee der Sequenz.
- Man verwendet Bezeichner, um die Objekte anzusprechen. Idee der Referenz.
- Wenn man wiederholt das gleiche tun will, kann man sich mit der Idee der Schleife Arbeit sparen.
- Objekte haben Attribute und Methoden. Diese können interaktiv aufgerufen und modifiziert werden.

3.2 Beispiel einer objektorientierten Modellierung

Es bietet sich auch die Möglichkeit, objektorientierte Modellierung genetisch einzuführen, indem man die folgenden verbundenen Schritte geht:

3.2.1 Modellierung eines Schneemanns

Ein Schneemann soll modelliert werden. Dazu muss ein gedankliches Modell geschaffen werden, das den Schneemann aus einfachen Grundkörpern zusammen setzt, und das wird dann implementiert:

```
from visual import *
S1b1=sphere(pos=(0,0,0),radius=2.4,color=color.white)
S1b2=sphere(pos=(0,2,0),radius=1.4,color=color.white)
S1b3=sphere(pos=(0,3.5,0),radius=0.9,color=color.white)
S1b4=sphere(pos=(–0.4,3.8,0.8),radius=0.2,color=color.black)
S1b5=sphere(pos=(0.4,3.8,0.8),radius=0.2,color=color.black)
S1b6=cone(pos=(0,3.5,0.8),axis=(0,0,1),radius=0.3,color=(0.5,1,0))
```

3.2.2 Konstruktion identischer Schneemänner

Das erzeugt einen Schneemann. Was aber, wenn man einen zweiten haben will? Man müsste den ganzen Code kopieren und die Koordinaten anpassen. Dass das mühsam und fehleranfällig ist, erleben die Schüler am eigenen Leibe, wenn sie es durchführen. Nach dieser Erfahrung sind sie bereit für das Lösungsmittel der Informatik: Prozedurale Abstraktion:

```
from visual import *
def Schneemann(p):
    S1b1=sphere(pos=vector(0,0,0)+p,radius=2.4,color=color.white)
    S1b2=sphere(pos=vector(0,2,0)+p,radius=1.4,color=color.white)
    # … analog wie oben weiter
    S1b6=cone(pos=vector(0,3.5,0.8)+p, axis=(0,0,1),radius=0.3,
    color=(0.5,1,0))
```

[2] *http://www.f-lohmueller.de/links/index_sd.htm*
[3] *http://www.lehrer-online.de/povray.php*

```
Schneemann((–2,0,0))
Schneemann((0,0,5))

for i in range(10): Schneemann((2*i–10,0,0))
```

3.2.3 Verschieben der Schneemänner

Eine solche Funktion, die ein Objekt konstruiert, könnte man mit vollem Recht Konstruktor nennen. Perfekt ist diese Lösung aber noch nicht: Man kann die so erzeugten Objekte nämlich nicht wie die Basisobjekte einfach verschieben. Es gibt verschiedene Strategien der Abhilfe: Der Konstruktor könnte eine Liste der Basisobjekte zurückgeben, und zum Verschieben muss man dann diese durchlaufen. Eine andere besteht darin, eine lokale Funktion zum Verschieben zu definieren und zurückzugeben. Das ist ein Beispiel für eine Methode.

```
from visual import *
import time
def Schneemann(p):
  S1b1=sphere(pos=vector(0,0,0)+p,radius=2.4,color=color.white)
  # …. Weiter wie oben
def setzePos(P):
  S1b1.pos=vector(0,0,0)+P
  S1b2.pos=vector(0,2,0)+P
  # u. s. w.
  S1b6.pos=vector(0,3.5,0.8)+P
  return setzePos

Schneemann((–2,0,0))
S3positioniere=Schneemann((0,0,5))

for i in range(10):
  time.sleep(0.5)
  S3positioniere((2*i–10,0,0))
```

Visual Python bietet aber auch die Möglichkeit, Objekte in ein Frame zu setzen und dann gemeinsam zu bewegen, indem man das Frame bewegt. In Bruchstücken sieht diese Lösung so aus:

```
def Schneemann(p):
  f=frame()
  sphere(frame=f,pos=vector(0,0,0)+p,radius=2.4,color=color.white)
  # …
  return f

S=Schneemann((0,0,5))
S.pos=(2*i–10,0,0)
```

3.2.4 Ein Schneemann mit Hut: Vererbung

Das ist schon recht elegant. Einziges Problem damit: Evtl. sollen nicht alle Schneemänner gleich sein, sondern es soll welche mit Hut geben. Das lässt sich am besten mit Objektorientierung und Vererbung in den Griff bekommen. Auch dazu gibt es viele Wege, wir wählen denjenigen, den Python dafür vorsieht:

```
from visual import *
class Schneemann(frame):
  def __init__(self,p):
    frame.__init__(self)
    S1b1=sphere(frame=self,pos=vector(0,0,0)+p,radius=2.4,
    color=color.white)
    S1b2=sphere(frame=self,pos=vector(0,2,0)+p,radius=1.4,
    color=color.white)
    #usw.
    S1b6=cone(frame=self,pos=vector(0,3.5,0.8)+p,axis=(0,0,1),
    radius=0.3,color=(0.5,1,0))

class SchneemannMitHut(Schneemann):
  def __init__(self,p):
    Schneemann.__init__(self,p)
    cylinder(frame=self,pos=vector(0,4,0)+p,radius=1.8,
    axis=(0,0.2,0),color=color.blue)
    cylinder(frame=self,pos=vector(0,4,0)+p,radius=0.8,
    axis=(0,2,0),color=color.blue)

S1=Schneemann((–2,0,0))
S2=SchneemannMitHut((2,0,0))
S3=Schneemann((0,0,5))
```

Zusammenfassend lässt sich sagen, dass hier ein Anwendungsgebiet vorliegt, in dem die Strukturierungsmöglichkeiten der objektorientierten Programmierung gut zur Geltung kommen und auch für Schüler erfahrbar einen Gewinn bringen. Es soll aber auch betont werden, dass man nie den ganzen gezeigten Weg von den Anfängen der Programmierung bis zur Objektorientierung an einem Stück gehen wird.

Wenn die Schüler anfangs noch die prozedurale Abstraktion kennen lernen müssen, wird man nicht in einem Rutsch bis zum Ende gehen, und wenn sie bereits weiter sind, wird der Anfang weniger ausführlich gestaltet.

Visual Python kümmert sich darum, dass die Objekte perspektivisch korrekt auf dem Bildschirm dargestellt werden. Die nötigen Zentralprojektionen sind mit der Vektorrechnung der gymnasialen Oberstufe kein Problem. (In OLDENBURG (2006) wird dies und der umgekehrte Weg – aus Digitalfotos 3D-Koordinaten bestimmen – elementar durchgeführt.) Da dies aber überwiegend Mathematik ist, verfolgen wir diesen Weg hier nicht weiter, sondern weisen darauf hin, dass man mit der einfachen Zeile

```
scene.stereo='redcyan'
```

die Szene so gerendert wird, dass sie mit Rot-Cyan-Anaglyphenbrillen räumlich erscheint. Auch wer über besserer Hardware (Shutterbrillen o. ä.) verfügt, wird von Visual Python bedacht. Damit ist man nicht nur mitten in der Physik (Farbfilter), sondern auch in der Biologie unserer Augen und dem Sehprozess.

3.3 Zugängliche Themen mit Visual Python

Um das Feld der mit Visual Python einfach zugänglichen Themen anzureißen, kommen hier einige Beispiele:

3.3.1 Zufallsgraphen

Für die Mathematik ist es ganz praktisch, sich übliche Objekte zu definieren:

```
def Punkt(x,y,z):
  return sphere(pos=(x,y,z),radius=0.3,color=color.red)
def Strecke(A,B): # zwischen Punkten A und B
  return cylinder(pos=A.pos,axis=B.pos-A.pos,radius=0.1,
  color=color.blue)
```

Damit kann man etwa vollständige Zufallsgraphen erzeugen (Abb. 3):

```
n=20
Punkte=[]
for i in range(n):
  [x,y,z]=[randint(–10,10),randint(–10,10),randint(–10,10)]
  Punkte.append(Punkt(x,y,z))
for A in Punkte:
  for B in Punkte: Strecke(A,B)
```

Abb. 3. Ein Zufallsgraph aus 20 Punkten

3.3.2 Animationen

Aus Listen von Punkten lassen sich Kurven zusammenbasteln und damit parametrische Plots von Kurven erzeugen (Abb. 4).

```
from visual import *
from math import *
import time

Punkte=[]
for alpha in range(1800):
    [x,y,z]=[5*sin(alpha/180.0*pi), 5*cos(alpha/180.0*pi),
    alpha/360.0–5]
    Punkte.append(vector(x,y,z))

curve(pos=Punkte, radius=0.05)
# Jetzt noch einen animierten Punkt entlang der Kurve schieben
ball=sphere(radius=0.5,color=color.red)
for p in Punkte:
    ball.pos=p
    time.sleep(0.05)
```

Neben der Steuerung von Simulationen über den sleep-Befehl bietet VPython noch die Möglichkeit zu bestimmen, dass eine Schleife höchstens n-mal pro Sekunde durchlaufen werden soll. Die dafür nötige Wartezeit wird dann so angepasst, dass die eigentliche Rechenarbeit plus die Wartezeit gerade den entsprechenden Sekundenbruchteil gibt. Das geht so:

```
>>> i=100
>>> while i>0:
        print i
        i=i–1
        rate(5) # Nur 5 Schleifendurchläufe pro Sekunde
```

3.3.3 Simulationen

Bei NEWTON kam die Motivation zur Entwicklung der Differenzialrechnung aus der Physik. Differenzialgleichungen haben sich als das erfolgreichste Beschreibungsinstrument für Bewegungsvorgänge aller Art erwiesen. Die Gleichung $F = ma$ ist eine Differenzialgleichung, denn die Beschleunigung ist die zweite Ableitung der Position $a(t) = x''(t)$. Die erste Ableitung des Ortes ist die Geschwindigkeit: $v(t) = x'(t)$, also $a(t) = v'(t)$. Mit der Grundvorstellung der lokalen linearen Näherung kann man das auflösen:

$$x(t + \Delta t) = x(t) + v(t) \cdot \Delta t,\ v(t + \Delta t) = v(t) + a(t) \cdot \Delta t\,.$$

Für vektorielle Größen gilt genau das Gleiche, wobei für die Beschleunigung gleich Kraft/Masse eingesetzt wird:

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \frac{1}{m} \cdot \vec{F}(\vec{x}, \vec{v}, t) \cdot \Delta t$$

$$\vec{x}(t + \Delta t) = \vec{x}(t) + \vec{v}(t) \cdot \Delta t$$

Damit kann man z. B. einen springenden Ball programmieren:

```
from visual import *

boden = box(length=8, height=0.5, width=8, color=color.blue)
ball = sphere(pos=(0,4,0), color=color.red)
ball.v = vector(0,–1,0) # Ballgeschwindigkeit : Anfangswert
ball.m=5                # Ballmasse
unten=vector(0,–1,0)    # Richtung der Schwerkraft
g=9.81                  # Erdbeschleunigung
dt = 0.01               # Zeitschritt

while true 1:
    rate(100)
    kraft=g*unten*ball.m
    ball.v+= kraft/ball.m*dt
    ball.pos+= ball.v*dt
    if ball.y < 1: ball.v.y = –ball.v.y
```

Die gleiche Technik reicht auch aus, die Bahn des Mondes um die Erde zu bestimmen.

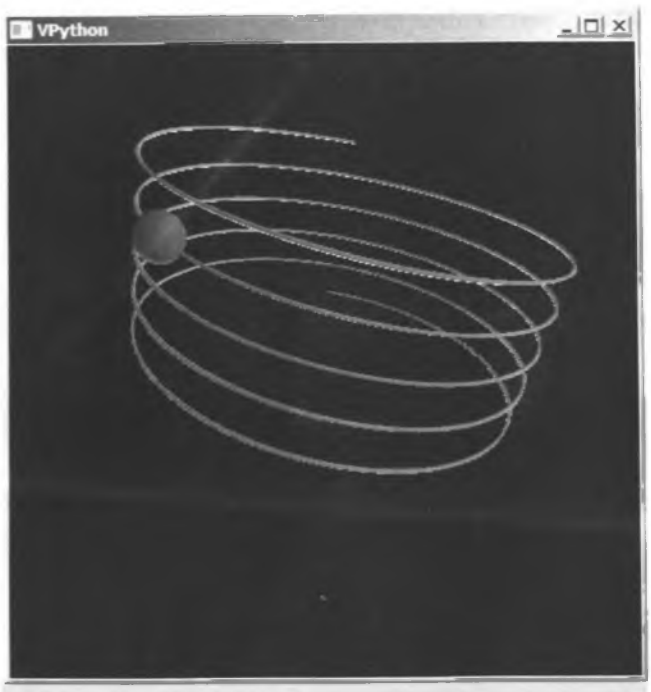

Abb. 4. Kugel auf einer Helix

3.3.4 Spiele

VPython erlaubt festzustellen, auf welches Objekt die Maus zeigt – im Raum ist das gar nicht so einfach und daher eine willkommene Arbeitserleichterung. Es ist deswegen leicht, ein Programm zu schreiben, mit dem man ein Objekt durch den virtuellen Raum navigiert. Ob daraus ein Spiel oder ein dynamisches Raumgeometrieprogramm wird, ist dann die nächste Frage. Beides ist realisierbar.

3.4 Didaktisches Fazit zu Visual Python

Die im ersten Teil zu PovRay genannten Bildungsziele lassen sich weitgehend auch für Visual Python anführen, allerdings mit anderer Gewichtung. Beispielsweise erreicht man auch mit viel Mühe keine photorealistische Darstellung, so dass die Motivation, über die Problematik gänzlich künstlicher »Photos« nachzudenken, deutlich geringer ist. Trotzdem bietet auch Visual Python zu allen prozess- wie inhaltsbezogenen Kompetenzbereichen der GI-Bildungsstandards Anknüpfungspunkte, insbesondere auch für den Kompetenzbereich »Informatik, Mensch und Gesellschaft«: Schüler können einen exemplarischen Einblick darin gewinnen, wie virtuelle Welten erzeugt werden, wie Vorgänge simuliert werden und wie (z.B mittels Anaglyphenbrillen), das Gehirn getäuscht werden kann.

»Begründen und Bewerten«: Der obige Weg zu den Konzepten der OOP im Kontext der 3D-Computergrafik hat gezeigt, dass es sehr oft verschiedene Wege gibt, bestimmte Ziele zu erreichen. Diese können dann vergleichend bewertet werden. Bei Animationen und Simulationen kann man begründen und bewerten, ob/dass das Simulationsergebnis angemessen ist.
Im Vergleich zu PovRay gewinnt man die Interaktivität und kann so leichter die traditionellen Ziele der Modellierung zeitlicher Abläufe mit Algorithmen bedienen.

4 Schlusswort

Es stehen Freeware-Tools zur Verfügung, mit denen dem Wunsch von Schülern, 3D-Grafik zu programmieren, entsprochen werden kann. Man sollte sich aber genau überlegen, was man damit erreichen will. Dabei hilft – hoffentlich – dieser Beitrag.

Die zahlreichen Beziehungen zu Themen der Mathematik, aber auch der Physik und der Biologie sollten klar machen, dass Informatik im Spektrum der Naturwissenschaften gut aufgehoben ist und viele Fragen dieser Wissenschaften aufwerfen kann.

Literatur

Gesellschaft für Informatik (2008). Grundsätze und Standards für die Schulinformatik. *www.informatikstandards.de* (12.12.10)

RABEL, M. & OLDENBURG, R. (2009). *Erwartungen und Wertungen von Schülern und Studenten.* INFOS 2009

OLDENBURG, R. (2004). *3D-Grafiken und Animationen mit POV-Ray.* X-Lab Göttingen.

OLDENBURG, R. (2006). *Rekonstruktion von 3D-Koordinaten aus Bildern.* In: Istron Bd. 9, Hildesheim.

LOHMÜLLER, F. A. (2010). Beschreibungen und Beispiele zum Raytracer POV-Ray. *http://www.f-lohmueller.de/pov_tut/pov__ger.htm#basic*

GRASSMANN, H. (2009). Dreidimensionale Geometrie mit Povray. *www.mathematik.hu-berlin.de/~hgrass/povray.pdf* (4.12.10)

MÖLLER, B. (2008): Graphikprogrammierung. *http://www.informatik.uni-augsburg.de/lehrstuehle/dbis/pmi/lectures/ws0809/graphik-programmierung/script/grafikprog.pdf*

Prof. Dr. REINHARD OLDENBURG, oldenbur@math.uni-frankfurt.de, Goethe-Universität Frankfurt, Institut für Didaktik der Mathematik und Informatik, Robert-Mayer-Str. 10, 60325 Frankfurt, war Lehrer für Mathematik, Physik und Informatik an einem Gymnasium in Göttingen, bevor er in die Lehrerausbildung wechselte, wo er sich besonders mit dem realitätsorientierten Unterricht mit Computernutzung beschäftigt. Er vertritt außerdem die Informatik im Vorstand des Fördervereins.

Dr. JÜRGEN POLOCZEK, juergen.poloczek@auge.de, Goethe-Universität Frankfurt, Robert-Mayer-Str. 10, 60325 Frankfurt, ist Fachleiter für Informatik und Mathematik am Studienseminar Oberursel (Gym), Karl-Hermann-Flachstr. 15b, 61440 Oberursel, und ist an das Institut für Informatik teilabgeordnet.

Steinerbäume (auch dreidimensionale) im Experiment

GERD LIMPERG

Diese Notiz soll die Arbeit von C. RÜHENBECK (2010) ergänzen. Sie beschreibt physikalische Experimente zu isoperimetrischen Problemen.

MNU 64/5 (15.7.2011) Seiten 284–288, ISSN 0025-5866, © Verlag Klaus Seeberger, Neuss