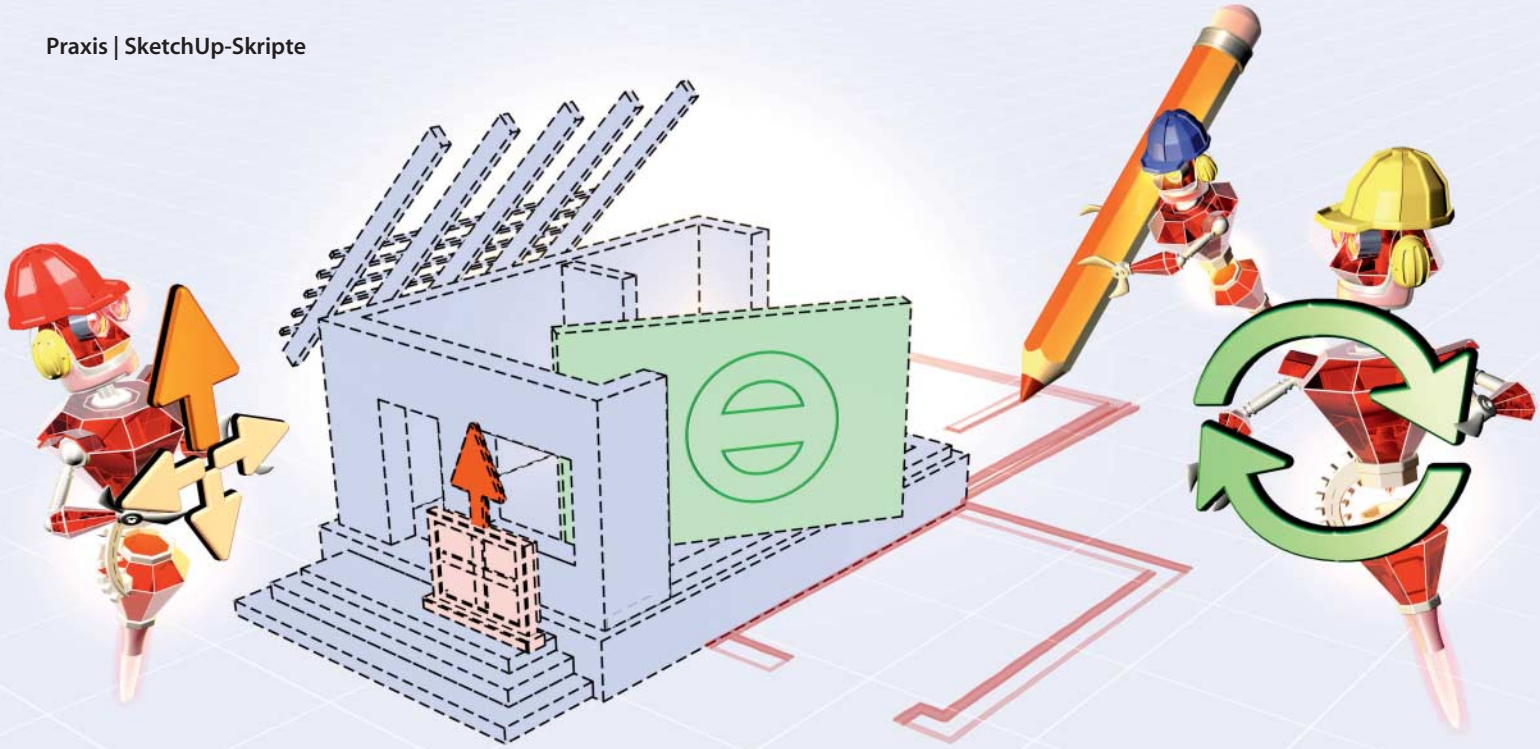


Jan Krutisch

Konstruieren mit Code

Eigene Werkzeuge für SketchUp programmieren

Googles kostenloses 3D-Zeichenprogramm bietet eine Ruby-Schnittstelle und ein umfangreiches API. Damit programmieren auch Ruby-Neulinge praktische Spezialwerkzeuge und lassen ihren Code Formen zaubern, die man in SketchUp von Hand nur mit Mühe hinbekommt.

Eigentlich verschenkt Google die Basisversion des 3D-Zeichenprogramms SketchUp, damit die Internet-Gemeinde den konzerneigenen Satellitenbildatlas Google Earth mit selbst zusammengeklickten, plastischen Modellen von Gebäuden anreichert. Allerdings taugt die Software darüber hinaus auch für etliche andere Zwecke, etwa für den Möbelentwurf oder um die Einrichtung der neuen Wohnung durchzuspielen. Comiczeichner modellieren mit SketchUp Vorlagen für Gegenstände, die schwer aus dem Kopf zu zeichnen sind. Und möchte man ein solches Objekt real in die Hand nehmen, eignen sich SketchUp-Modelle auch als Vorlagen zum 3D-Druck.

Für viele Vorhaben bringt SketchUp das notwendige Handwerkszeug von Haus aus mit. Man kann die Anwendung aber auch um eigene Hilfsmittel und Werkzeuge erweitern, die man in der Programmiersprache Ruby schreibt. Diese Interpretersprache wurde in den letzten Jahren vor allem durch das Open-Source-Web-Framework Ruby on Rails bekannt. Ruby ist eine sehr dynamische Sprache, die sich prima an spezielle Anwendungen anpassen lässt. Dabei entsteht schnell eine sogenannte Domain Specific Language (DSL), die auf den ersten Blick nicht viel mit Ruby zu tun zu haben scheint. Der Vorteil: Es ist keinesfalls nötig, Ruby erst einmal von der Pike auf zu lernen, um die Sprache für SketchUp-Werkzeuge einzusetzen. Sie erfüllt hier eher die Funktion einer speziellen Skriptsprache für den 3D-Zeichner.

Startklar

Beim Programmstart lädt SketchUp automatisch alle Skripte, die auf .rb für Ruby enden und die im Ordner namens Plugins oder einem seiner Unterordner liegen. Unter Windows findet man ihn im SketchUp-Installationsverzeichnis, typischerweise unter „C:/Program Files (x86)/Google/Google SketchUp 8/Plugins“, auf dem Mac unter „/Library/Application Support/Google SketchUp 8/SketchUp/Plugins“.

Für solche Skripte kann man neue Einträge ins Haupt- und Kontextmenü sowie die Toolbars von SketchUp packen. Beim Entwickeln und Debuggen eigener Funktionen ist es aber praktischer, seine Programme über die Ruby-Konsole zu laden. Diese öffnet man über das Fenstermenü. Die Konsole nimmt Ruby-Kommandos auch direkt entgegen. Probieren Sie doch mal eine Art „Hello World“-Programm aus und tippen Sie dazu in die Konsole:

```
UI.messagebox("Ruby rockt")
```

Wenn Sie dann Enter drücken, sollte die Nachricht im Programmfenster erscheinen.

Mehr als eine Zeile bekommt man in die Konsole nicht hinein, Berge kann man damit nicht versetzen. Für die folgenden Beispielprogramme brauchen Sie daher noch einen Texteditor, der mit Ruby-Code umgehen kann und möglichst das passende Syntax-Highlighting beherrscht – beispielsweise E, Notepad++ oder JEdit für Windows und TextMate oder Smultron auf dem Mac. Alle hier vorgestellten Skripte stehen unter dem c't-Link am Ende des Artikels zum Download.

Leerer Bauch

Fast jeder SketchUp-Neuling zeichnet als erstes ein Rechteck auf den Boden und zieht dieses dann zu einem Quader auf. Was mit den Zeichenwerkzeugen für die Maus klappt, geht auch mit Ruby. Legen Sie dazu eine Datei namens rechteck.rb in einem beliebigen Verzeichnis an und fügen Sie das Grundgerüst einer SketchUp-Ruby-Datei ein:

```
# Ruby-Grundgerüst
require 'sketchup'
def rechteck
end
```

Das Doppelkreuz # markiert eine Kommentarzeile, die der Ruby-Interpreter überspringt. Die zweite Zeile lädt die SketchUp-Bibliothek, sodass das Skript auf die Objekte und Klassen der Anwendung zugreifen kann. Dann folgt die Definition einer Methode von `def rechteck` bis `end`. Eine Methode ist so etwas wie ein selbst definierter Befehl. Im Beispiel ist die Methode namens `rechteck` zwar schon angelegt, sie tut allerdings noch nichts.

Um das Skript trotzdem schon mal auszuprobieren, könnte man es in den Plugins-Ordner kopieren und anschließend SketchUp

neu starten. Schneller geht es über die Ruby-Konsole: Hierzu tippen Sie dort den Befehl load ein, gefolgt vom Pfad zur Datei, den Sie mit Anführungszeichen umschließen, auf dem Mac zum Beispiel:

```
load "/Users/jan/Documents/Skipte/rechteck.rb"
```

Wer Windows benutzt, kann sich diesen Pfad aus der Explorer-Adressleiste kopieren, muss anschließend aber alle Backslashes \ in die Gegenrichtung / kippen:

```
load "C:/Skripte/rechteck.rb"
```

Hat das geklappt, meldet die Konsole lediglich true. Schließlich definiert der Code nur eine Methode, führt diese aber nicht aus. Hierfür muss man die Methode über die Konsole explizit aufrufen, indem man den Namen der Methode wie einen Befehl benutzt. Geben Sie dazu einfach mal rechteck ein:

```
rechteck
nil
```

In Ruby haben alle Methoden einen Rückgabewert – ist keiner über ein return-Statement explizit definiert, gibt es stets einen impliziten. Dabei steht nil für das Fehlen eines Werts; in anderen Programmiersprachen heißt das oft NULL. Hier bedeutet die Rückgabe von nil, dass die Methode rechteck funktioniert – sofern man das von einer Methode, die nichts tut, so sagen kann.

Es werde Rechteck

Um den eigenen Code auf das aktuell in SketchUp geöffnete 3D-Modell anwenden zu können, muss man sich mit den Klassen und Methoden des API befassen. Für den Fall, dass Sie noch nie mit einer objektorientierten Programmiersprache zu tun hatten und sich unter Begriffen wie Klasse, Typ und Objekt nichts Konkretes vorstellen können, stellt der Kasten diese Konzepte kompakt vor. Um unsere Beispielskripte zu benutzen und zu modifizieren, ist es aber nicht nötig, sämtliche Feinheiten der Arbeit mit der Programmierschnittstelle sofort zu durchschauen.

Die passende Füllung, damit die bisher leere Methode ein Rechteck zeichnet, sieht wie folgt aus:

```
def rechteck
  model = Sketchup.active_model
  entities = model.entities
  pt1 = [0, 0, 0]
  pt2 = [0, 9, 0]
  pt3 = [9, 9, 0]
  pt4 = [9, 0, 0]
  neue_flaeche = entities.add_face pt1, pt2, pt3, pt4
end
```

Hier bezeichnet Sketchup die Basis-Klasse des SketchUp-API. Die Methode active_model auf der SketchUp-Klasse liefert das aktuell in SketchUp geöffnete 3D-Modell zurück, als Anknüpfungspunkt für den eigenen Code. Mit der entities-Methode wiederum kommt man an die Sammlung aller Teile des Modells, die Objekte der Klasse Entity aus dem API sind. Fügt man dieser Sammlung per Code neue Teile hinzu, tauchen diese in der 3D-Darstellung von SketchUp auf.

Für ein Rechteck definiert man dessen vier Eckpunkte. Da SketchUps Welt ein dreidimensionaler Raum ist, muss man sich entscheiden, in welcher Ebene es liegen soll. Der Einfachheit halber spannt das Beispiel die Form zwischen der x- und der y-Achse auf, somit auf dem virtuellen Fußboden von SketchUp. Punkte kann man im Ruby-API entweder als Objekte vom Typ Point3d anlegen oder man schreibt wie im Beispiel die x-, y- und z-Koordinaten zwischen eckige Klammern und trennt sie mit Kommas, was lesbarer und einfacher ist. Diese Schreibweise definiert in Ruby ein sogenanntes Array, eine Liste von Werten, wie man sie in praktisch jeder Programmiersprache findet.

Wie man am Beispiel sieht, müssen Variablen in Ruby nicht erst deklariert werden. Man schreibt einfach einen Bezeichner wie pt1 hin und weist ihm einen Wert zu. Die Methode add_face erzeugt aus den Punkten schließlich eine Fläche und fügt sie in die Sammlung aller Teile des SketchUp-Modells ein.

Die neue Fläche ist eine Instanz der Klasse Face aus dem SketchUp-API. Beim Erzeugen muss man ihr mindestens drei Punkte mitgeben, die nicht auf einer Geraden liegen, um die Ebene zu definieren, in der die Fläche liegt. Man kann aber beliebig viele weitere Punkte hinzufügen, sofern sie alle in derselben Ebene liegen. Ein Rechteck beispielsweise entsteht aus vier Punkten, die seine Ecken festlegen.

Bei Methodenaufrufen darf man übergebene Parameter klammern, muss aber nicht. In unseren Beispielen haben wir optionale Klammern in der Regel weggelassen. Wenn Sie das übersichtlicher finden, können Sie aber gerne entities.add_face(pt1, pt2, pt3, pt4) schreiben; Ruby ist das gleich.

Laden Sie Ihre Datei noch einmal über die Ruby-Konsole in SketchUp. Mit der Cursor-hoch-Taste können Sie frühere Eingaben wieder aufrufen und sparen sich so die erneute Pfad-Tipperei. Führen Sie die Methode über den Befehl rechteck aus. Glückwünsche an die Eltern, es ist ein Quadrat!

Hoch hinaus

Durch Parametrisierung wird die Methode flexibler, sodass man darüber Rechtecke in beliebiger Größe zeichnen kann. Dazu ersetzt man die konkreten Punktkoordinaten durch Variablen und fügt der Methodendefinition die Parameter breite und tiefe hinzu:

```
def rechteck(breite, tiefe)
  modell = Sketchup.active_model
  entities = modell.entities
  pt1 = [0, 0, 0]
  pt2 = [0, tiefe, 0]
  pt3 = [breite, tiefe, 0]
  pt4 = [breite, 0, 0]
  neue_flaeche = entities.add_face pt1, pt2, pt3, pt4
end
```

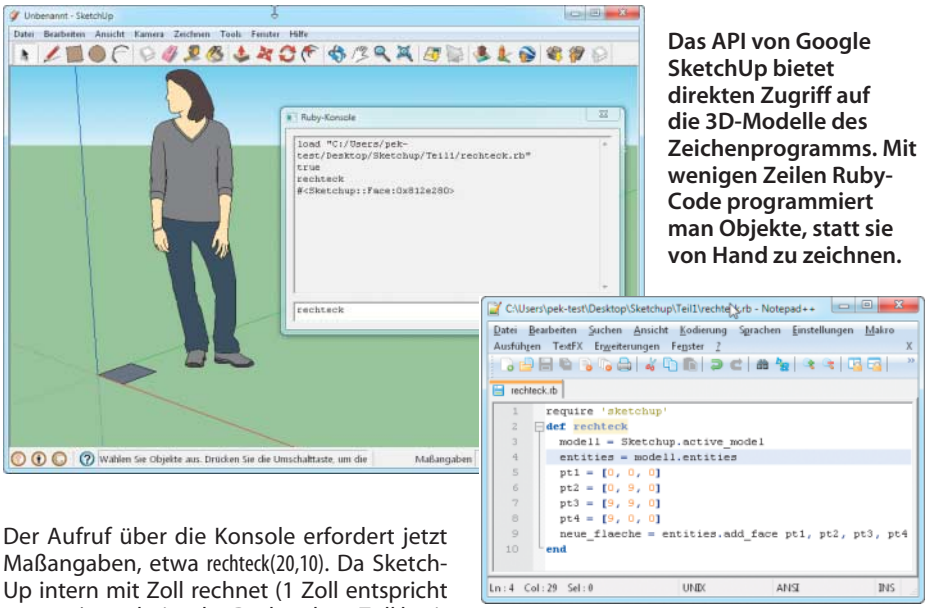
Klassen, Objekte, Methoden

Möchte man mit der Programmierschnittstelle (API, Application Programming Interface) von Googles kostenlosem 3D-Zeichenprogramm SketchUp arbeiten, muss man zur Programmiersprache Ruby greifen. Im Unterschied zu den Skriptsprachen vieler Anwendungen ist Ruby eine eigenständige objektorientierte Sprache.

Nach dem Paradigma der objektorientierten Programmierung besteht sowohl die reale Welt als auch deren Abbildung im Datenmodell im Computer aus lauter einzelnen **Objekten**, von denen jedes individuelle und genau festgelegte Eigenschaften hat. Objektorientiert gesehen sind beispielsweise alle Häuser einer Stadt jeweils eigene Objekte. Sie unterscheiden sich voneinander in ihren Eigenschaften wie der Zahl der Fenster und Stockwerke sowie der Adresse. Trotzdem gehören sie alle zum gleichen **Typ**, den man beispielsweise Gebäude nennen könnte: Ihnen allen ist gemeinsam, dass sie überhaupt Fenster und Stockwerke haben, deren Zahl man festhalten kann, und sich an einem festen Ort befinden, sodass sich eine Adresse zuordnen lässt.

Möchte man die Stadt jetzt in einem Programm in einer objektorientierten Sprache modellieren, programmiert man zunächst eine **Klasse**, die eine allgemeine Beschreibung von Objekten des Typs Gebäude festlegt. Die Klasse legt man so an, dass sie Variablen oder **Felder** enthält, die Informationen zur Anzahl von Fenstern und Stockwerken sowie Adressen aufnehmen. Die Klasse funktioniert wie ein leeres Formular, das man beliebig vervielfältigen und mit konkreten Werten füllen kann. Genau das passiert, wenn man im Programm Objekte einer Klasse erzeugt. Ein Objekt im Programm ist eine konkrete **Instanz** des Typs, der in der Klasse definiert wird.

Neben Datenfeldern für bestimmte Eigenschaften von Objekten eines Typs (wie der Fensteranzahl) definiert man in der Klasse auch **Methoden**, die man auf die Objekte anwenden kann. Methoden sind so etwas wie selbst geschriebene Befehle. Für Häuser kann man beispielsweise Methoden schreiben, um einem Haus einen neuen Bewohner hinzuzufügen oder sich die Namen aller Bewohner ausgeben zu lassen. Im ersten Fall gibt man der Methode beim Aufruf den Namen des Einziehenden als **Parameter** mit, im zweiten Fall erhält man die Namensliste als **Rückgabewert** der Methode.



Das API von Google SketchUp bietet direkten Zugriff auf die 3D-Modelle des Zeichenprogramms. Mit wenigen Zeilen Ruby-Code programmiert man Objekte, statt sie von Hand zu zeichnen.

Der Aufruf über die Konsole erfordert jetzt Maßangaben, etwa rechteck(20,10). Da SketchUp intern mit Zoll rechnet (1 Zoll entspricht 2,54 cm), erscheint das Rechteck 20 Zoll breit und 10 Zoll tief. Zwar kann man SketchUp auf metrische Angaben umstellen, das API rechnet intern aber mit Zoll. Wenn Sie die Maße Ihrer Rechtecke lieber in Zentimetern angeben wollen, benutzen Sie einfach die Methode Numeric.cm aus dem API:

```
pt3 = [breite.cm, tiefe.cm, 0]
```

Um in die dritte Dimension vorzustoßen, fügen Sie eine neue Methode in Ihre Datei ein, die aus dem Rechteck einen Quader macht:

```
def kasten(breite, tiefe, hoehe)
  flaeche = rechteck(breite, tiefe)
  flaeche.pushpull(hoehe)
end
```

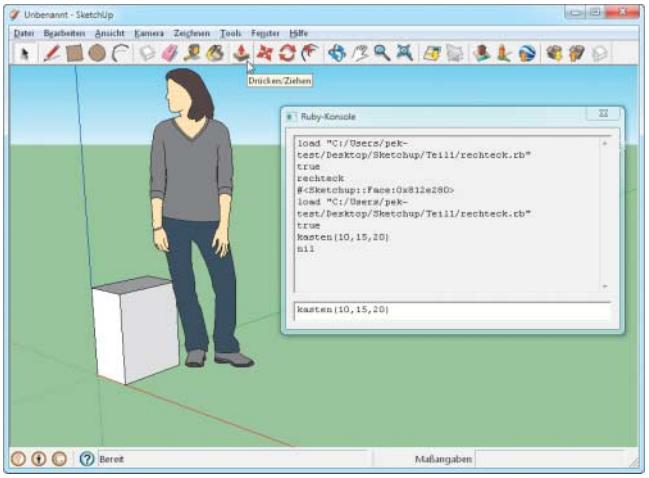
Die Methode pushpull ist das Code-Äquivalent zum Drücken/Ziehen-Werkzeug in SketchUp: Damit kann man aus Flächen dreidimensionale Körper herausziehen. Wenn Sie die Methode ausprobieren, wird Ihnen auffallen, dass der Quader in den Boden hineinwächst. Das liegt an der vorgegebenen Orientierung der Fläche. Es gibt zwei Wege, das zu beheben: Entweder übergibt man an pushpull einfach eine negative Zahl. Oder man dreht die Fläche vorher um. Das erledigt eine eingefügte Zeile flaeche.reverse! vor dem pushpull-Aufruf.

Das beschriebene Skript finden Sie auch unter dem c't-Link am Ende des Artikels mit dem Dateinamen rechteck_und_kasten.rb.

Im Rahmen …

Will man nur einen Quader zeichnen, spart ein Ruby-Skript nicht wirklich Zeit. Anders sieht das bereits bei einem nur moderat komplizierten Objekt wie einem Rahmen aus. Kann man hierfür einfach vier Zahlen für Breite, Höhe, Dicke und Tiefe angeben, geht das schneller, als wenn man mit der Maus und dem Versatz-Werkzeug einen Rahmen in der gewünschten Dicke um ein Ausgangsrechteck zeichnet und den anschließend auf Höhe zieht.

Der Rahmen wird zunächst zweidimensional angelegt, als Rechteck mit einem ebenfalls rechteckigen Loch darin. Dazu legt man erst die äußere und dann die innere Form als Rechtecke an und stanzt das innere aus dem äußeren aus. Mit der Maus würde man die Innenform mit dem Auswahlpfeil markieren und dann über Bearbeiten/Löschen das Loch öffnen. Anschließend gruppiert man den zweidimensionalen Rahmen, um im Folgenden einfacher mit ihm weiterarbeiten zu können.



Alles, was mit den Werkzeugen aus den Iconleisten von SketchUp geht, funktioniert auch mit Ruby-Code: Hier zieht die Methode pushpull aus dem Rechteck auf dem Boden einen dreidimensionalen Quader auf – genau wie das Drücken/-Ziehen-Werkzeug.

Baut man den Rahmen statt mit Handwerkzeugen über ein Skript, geht man prinzipiell ähnlich vor. Allerdings legt man hier zuerst die neue Gruppe an, mittels:

```
gruppe = modell.entities.add_group
```

Diese besteht aus einer Sammlung von Objekten (Entities), der man die Rechtecke direkt hinzufügt:

```
aussenflaeche = gruppe.entities.add_face(
    [0,0,0], [10,0,0], [10,10,0], [0,10,0])
innenflaeche = gruppe.entities.add_face(
    [2,2,0], [8,2,0], [8,8,0], [2,8,0])
```

Beide Flächen gehören jetzt zur selben Gruppe. Die folgenden Zeilen stanzen zunächst die Innenfläche aus der Außenfläche aus und sorgen dann für die gewünschte Dicke des Rahmens von zwei Zoll:

```
innenflaeche.erase!
aussenflaeche.pushpull -2
```

Das komplette Rahmen-Skript finden Sie online unter dem Dateinamen rahmen.rb.

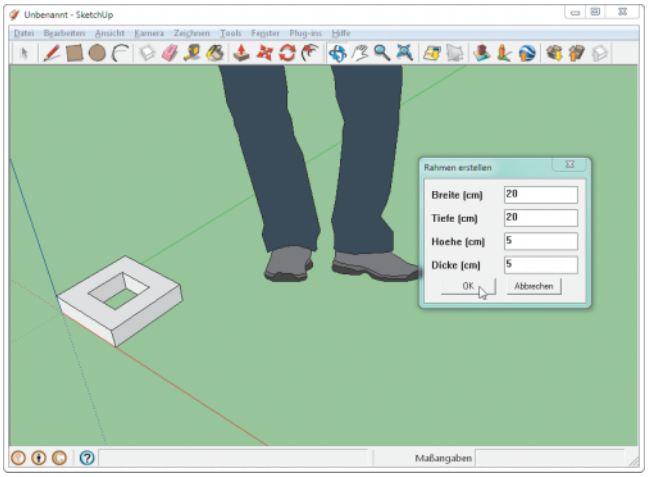
… und im Dialog

Der Aufruf über die Konsole eignet sich prima fürs Testen. Im produktiven Einsatz möchte man seine selbst programmierten Helferlein aber übers Menü aufrufen und am besten auch noch alle Parameter über ein Fenster mit Eingabefeldern festlegen. Deshalb haben wir dem Skript in der Datei rahmen.rb noch eine zweite Methode namens rahmen_mit_dialog hinzugefügt, die die gewünschten Werte für Breite, Höhe, Tiefe und Dicke des Rahmens über eine sogenannte Inputbox vom Benutzer erfragt und anschließend die vorige Methode aufruft:

```
def rahmen_mit_dialog
  namen = ['Breite (cm)', 'Tiefe (cm)',
      'Hoehe (cm)', 'Dicke (cm)']
  werte = [20, 20, 5, 5]
  breite, tiefe, hoehe, dicke =
      UI.inputbox namen, werte, "Rahmen erstellen"
  rahmen(breite.cm, tiefe.cm, hoehe.cm, dicke.cm)
end
```

Die Methode inputbox erwartet drei oder vier Parameter: An erster Stelle muss ein Array mit den Beschriftungen für die Eingabefelder stehen, die der Dialog in SketchUp später zeigen soll. Als zweiter Parameter dient ein Array mit Standardwerten für die Eingabefelder. Der dritte Parameter ist entweder der Titel für die Dialogbox wie hier oder ein Array mit Werten, die aus den Eingabefeldern Dropdown-Listen machen. Ein Beispiel dafür sehen Sie im Code in der Datei kopie.rb, die später noch ausführlicher beschrieben wird. In so einem Fall braucht die Methode noch einen vierten Parameter, der dann für den Titel der Box reserviert ist.

Die Methode inputbox gibt ein Array zurück, dessen Elemente in derselben Reihenfolge stehen wie die beiden ersten Parameter-Arrays beim Erzeugen der Inputbox. Die zuvor beschriebene Methode rahmen wiederum



Statt als Parameter über den Konsolenaufruf kann man seiner Methode die notwendigen Maße auch über eine Dialogbox übergeben.

möchte ihre Parameter in genau derselben Reihenfolge haben. Verzichtet man darauf, die Parameter wie hier noch in Zentimeter umzurechnen, könnte man zu einem kleinen Ruby-Trick greifen und den sogenannten Splat-Operator * das von der Inputbox zurückgelieferte Array in Parameter aufteilen lassen. Die beiden letzten Zeilen des Methodenrumpfes lauteten dann:

```
resultat = UI.inputbox namen, werte, "Rahmen erstellen"
rahmen(*resultat)
```

Jetzt fehlt nur noch ein Menüeintrag. Den erzeugt man einfach über die UI.menu-Methode:

```
UI.menu("Plug-Ins").add_item("Rahmen erstellen") do
    rahmen_mit_dialog
end
```

Leider hat die Sache mit dem Menüeintrag einen Haken, wenn man etwas am Code ändert und dann die Datei neu lädt: Dabei wird auch der Eintrag nochmals angelegt, sodass das Menü am Ende aus einer ellenlangen Liste von „Rahmen erstellen“-Einträgen besteht. Es gibt leider auch keine Funktion, um Menüeinträge im laufenden Betrieb zu löschen.

Ein Neustart von SketchUp behebt das Problem. Darüber hinaus gibt es einen Trick, der das Problem abmildert und der ein bisschen an die #ifdef-Konstrukte aus C-Programmen erinnert: Man kann prüfen, ob eine Datei bereits einmal geladen wurde und den Menüeintrag nur beim erstem Mal hinzufügen:

```
unless file_loaded? File.basename(__FILE__)
  UI.menu("Plug-Ins").add_item("Rahmen erstellen") do
    rahmen_mit_dialog
  end
end
file_loaded File.basename(__FILE__)
```

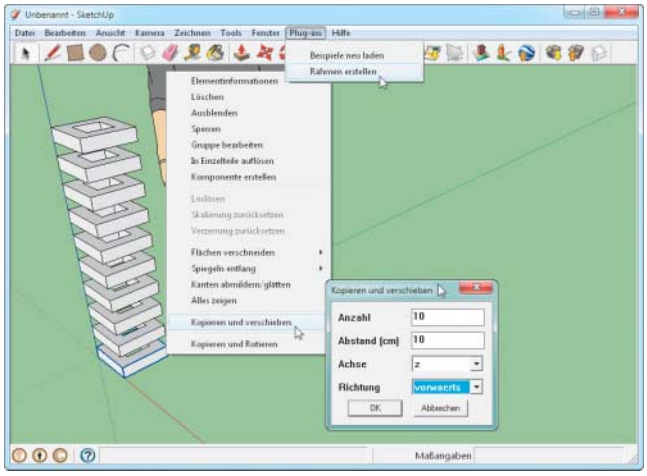
So kann man am Code weiterarbeiten und müllt dennoch das Menü nicht voll. Ein SketchUp-Neustart wird damit nur noch fällig, wenn man am oben gezeigten Code für den Menüeintrag selbst etwas ändert, zum Beispiel den anzuzeigenden Text.

Recycling

Mit Ruby-Code kann man nicht nur neue Objekte erzeugen, man kommt auch an Teile eines Modells heran, das man von Hand gezeichnet oder aus Googles 3D-Galerie geladen hat. Als erste Fingerübung dazu eignet sich ein Skript, das ein beliebiges Objekt entlang einer der drei Koordinatenachsen in frei wählbarem Abstand vervielfältigt.

Um die Bearbeitung zu vereinfachen, gruppiert es zunächst die ausgewählten Objekte. Das gelingt in einer Zeile:

```
modell.entities.add_group(modell.selection)
```



Das selbst geschriebene Rahmenwerkzeug nistet sich im neu angelegten Plug-ins-Menü ein, die Funktion fürs Vervielfältigen erreicht man über das Kontextmenü.

Darauf ruft das Skript die copy-Methode aus dem API auf, um das zu einer Gruppe zusammengefasste Objekt zu vervielfältigen. Danach gilt es, die Kopie noch zu verschieben.

Hierzu bietet das API die Methode transform!. Welche Transformation man konkret vorhat, muss man zuvor definieren, indem man ein Objekt vom Typ Transformation erzeugt. Diese Klasse stellt diverse Konstruktoren (Methoden zum Anlegen neuer Objekte) etwa für Skalierungen, Rotationen und Verschiebungen (Translationen) zur Verfügung. Das passende Objekt für eine Verschiebung um 10 Zoll entlang der x-Achse erzeugt man durch:

```
transformation = Geom::Transformation.translation([10,0,0])
```

Entlang welcher Achse verschoben wird, teilt man der Methode kopieren_und_verschieben über einen Parameter names achse mit. Der Befehl case unterscheidet die drei möglichen Fälle und bestimmt, wie die Translation definiert wird. Zusätzlich übergibt man man noch die Richtung als 1 oder –1:

```
verschiebung = case(achse)
when :x
  [abstand * i * richtung, 0, 0]
when :y
  [0, abstand * i * richtung, 0]
when :z
  [0, 0, abstand * i * richtung]
end
```

Als Ergebnis der Fallunterscheidung enthält die Variable translation ein Array mit drei Koordinaten, aus der das Skript eine Translation erzeugt und damit die Kopie der ausgewählten Objekte transformieren kann:

```
transformation =
    Geom::Transformation.translation(verschiebung)
kopie.transform!(transformation)
```

Dies ist grob das Gerüst des Skripts. Die Luxus-Version mit dem Namen kopie.rb finden Sie online. Hier kapselt die Methode gruppe_aus_auswahl ein paar Sonderfälle. Sie prüft etwa, ob es sich beim aktuell ausgewählten Objekt bereits um eine Gruppe handelt. Im eigentlichen Kopierskript werden Sie diese Methode vergeblich suchen – sie steckt in der Datei werkzeuge.rb im Unterverzeichnis lib. Ins Skript kopie.rb wird diese Datei über die Zeile require 'werkzeuge' eingebunden.

Die Auslagerung in eine andere Datei führt allerdings dazu, dass man dieses Skript nicht mehr über den load-Befehl und die Konsole laden kann. Öffnen Sie stattdessen das Skript lader_teil_1.rb aus unserem Beispielpaket im Editor und fügen Sie als Wert für die Variable this_path den Pfad zu dem Verzeichnis ein, in dem Sie die Ruby-Skripte abgelegt haben. Anschließend speichern Sie lader_teil_1.rb und verschieben die Datei in den Plugins-Ordner von SketchUp.

Wenn Sie SketchUp neu starten, wird dieses Skript automatisch ausgeführt und lädt alle Beispiele aus dem bezeichneten Ordner; die Konsole brauchen Sie dann nicht mehr. Gleichzeitig erweitert lader_teil_1.rb die Menüs von SketchUp: So taucht im Haupt-

menü der Eintrag „Plug-ins“ auf, der die handgestrickte Hilfsfunktion „Beispiele neu laden“ enthält. Hat man den Code seiner Ruby-Skripte geändert, lädt man sie darüber mit einem Klick neu ins Programm, ohne SketchUp beenden zu müssen. Unser Werkzeug „Kopieren und verschieben“ erreicht man über einen Eintrag im Kontextmenü von SketchUp. Solche Einträge funktionieren etwas anders als jene im normalen Menü, da man hier noch Vorbedingungen prüfen kann. Im Beispiel testen wir, ob überhaupt etwas ausgewählt ist. Zuvor setzen wir noch die Abfrage file_loaded?, sonst würde jedes Laden der Datei einen neuen Kontextmenü-Eintrag hinzufügen:

```
unless file_loaded? File.basename(__FILE__)
  UI.add_context_menu_handler do |menu|
    if model.selection.length > 0
      menu.add_separator
      menu.add_item("Kopieren und verschieben") {
          kopieren_und_verschieben_mit_dialog }
    end
  end
end
file_loaded File.basename(__FILE__)
```

Sternstunde

Mit der vorgestellten Methode kann man schon viele Dinge anstellen – Säulenreihen bauen, Alleen pflanzen oder Menschenschlangen aufreihen. Noch interessanter wird es, wenn man die Gegenstände beim Verschieben rotiert, um etwa aus einem einzelnen Zylinder einen Sternmotor oder aus einer Schaufel eine Turbine zusammenzubauen.

Auch für Rotationen gibt es einen eigenen Konstruktor in der Klasse Geom::Transformation:

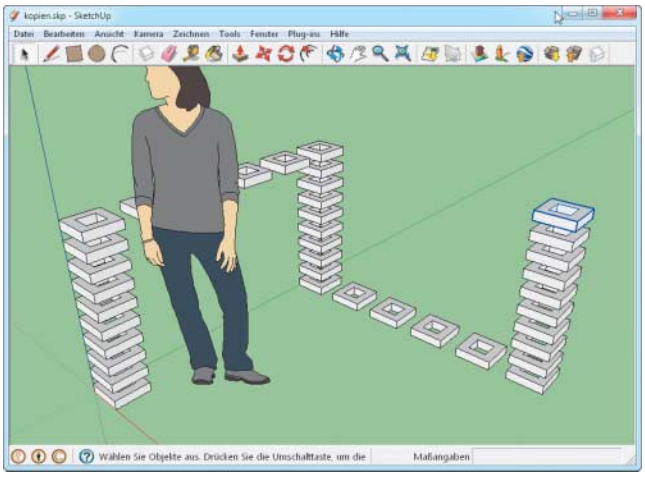
```
trans = Geom::Transformation.rotation(p, a, w)
```

Oberflächlich sieht das ganz einfach aus, im Detail wird es allerdings interessant: Die Parameter w und a geben den Winkel und den Achsenvektor der Rotation an. Beschränkt man sich bei letzterem zunächst auf die Grundachsen, ist er entweder [1,0,0], [0,1,0] oder [0,0,1]. Den Winkel muss man im Bogenmaß angeben, 360 Grad entsprechen somit 2 π.

Das erste Argument der Methode ist besonders interessant: Über den Punkt p gibt man den Rotationsmittelpunkt an. Dieser lässt sich beliebig platzieren, er muss insbesondere nicht innerhalb des Ausgangsobjekts liegen. In unserem Beispiel rotationskopie.rb nutzen wir diese Eigenschaft, um die Kopien von Objekten exzentrisch zu verteilen, wie man das etwa beim Rotorenbau braucht.

Die Berechnung muss dabei berücksichtigen, dass die Ursprungskoordinate nicht die Mitte der Gruppe ist, die erzeugt wird. Der Kern der Operation sieht so aus:

```
(anzahl).times do |i|
  kopie = gruppe.copy
  kopie.transform! Geom::Transformation.rotation(punkt,
      achsenvektor, schrittweite * i)
  kopien << kopie
end
```



Jede aufgereihte Kopie ist ein eigenständiges Objekt, das sich erneut mit unserem Vervielfältigungswerkzeug kopieren lässt – auch mit abweichendem Abstand und in neuer Richtung.

Der Winkel zwischen zwei benachbarten Rotationskopien (hier als schrittweite bezeichnet) berechnet sich daraus, wie viele Kopien insgesamt angelegt werden sollen und in wie viele Teile der Vollkreis somit unterteilt werden soll:

```
schrittweite = 2 * Math::PI / anzahl
```

Kommando zurück

Führt man in SketchUp ein Ruby-Skript aus und ruft danach Bearbeiten/Rückgängig auf oder drückt Strg+Z, wird nur der letzte Befehl des Skripts rückgängig gemacht. Das kann zwar praktisch sein, da man auf diese Weise eigene Programme quasi rückwärts debuggen kann. Wenn Sie allerdings eine gut getestete Operation zurücknehmen wollen und die gerade ein Objekt 200-mal vervielfältigt hat, macht das dann doch keinen Spaß mehr.

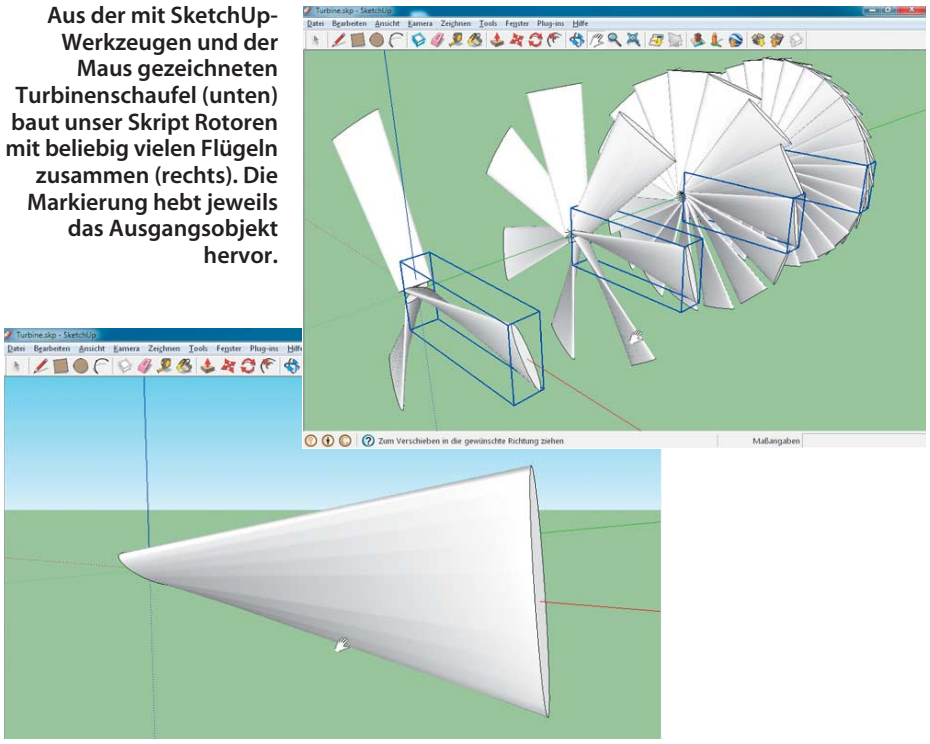
Will man das komplette Skript in einem Schritt rückgängig machen, gibt es im SketchUp-API ein sehr schönes Konstrukt: Sie können alle Operationen, die Sie in einem Schritt zusammenfassen wollen, mit einem speziellen Methoden-Paar umschließen:

```
modell.start_operation "Kopieren und rotieren"
...
modell.commit_operation
```

Der start_operation kann man sogar einen Namen mitgeben, der den „Rückgängig“-Eintrag im SketchUp-Menü ergänzt.

Mit den bisher beschriebenen Konstrukten und Kniffen lassen sich bereits nützliche Skripte programmieren, die viel Handarbeit ersparen. Wie Sie rein mit Code beliebig fein gerundete Kugeln und Gewinde im kostenlosen 3D-Zeichner hervorzaubern und dabei auch noch Ihre Funktionen sauber in eigene Klassen kapseln, lesen Sie in der nächsten c’t. (pek)

www.ct.de/1102158



Aus der mit SketchUp-Werkzeugen und der Maus gezeichneten Turbinenschaufel (unten) baut unser Skript Rotoren mit beliebig vielen Flügeln zusammen (rechts). Die Markierung hebt jeweils das Ausgangsobjekt hervor.