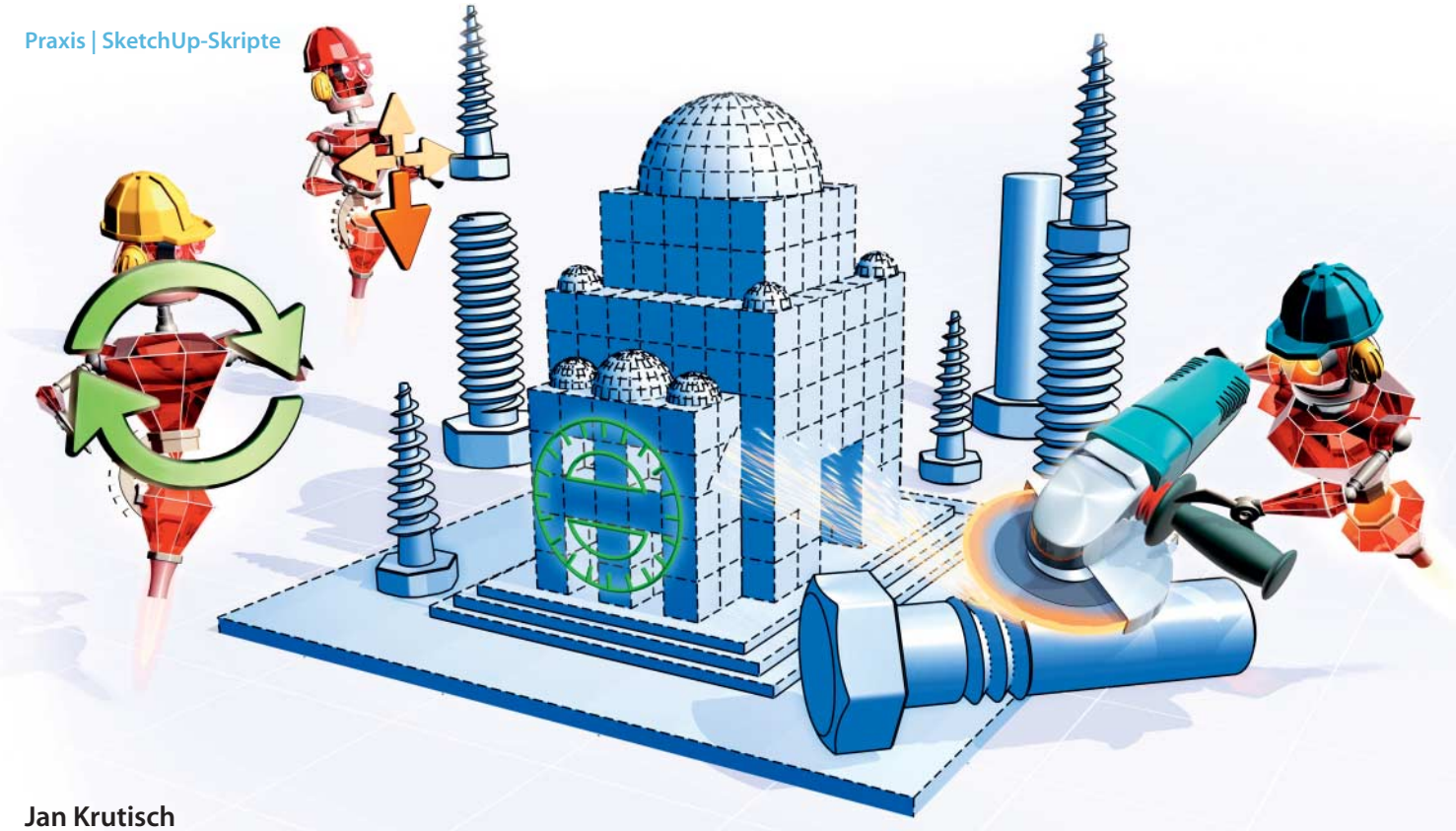


Jan Krutisch

# Konstruieren mit Klasse

## Eigene Werkzeuge für SketchUp programmieren, Teil 2

**Selbst geschriebene Ruby-Skripte nehmen einem bei der dreidimensionalen Konstruktion nicht nur lästige Routineaufgaben ab, sie erzeugen auch komplexe Formen wie Gewinde, die man von Hand praktisch nicht hinbekommt.**

Wer in Googles kostenlosem 3D-Zeichenprogramm SketchUp plastische Modelle bauen will, muss sich keinesfalls mit den vorkonfektionierten Werkzeugen begnügen: Dank der Ruby-Schnittstelle und des passenden API schreibt man vermisste Makros und Assistenten kurzerhand selbst. Den Einstieg haben wir in der vergangenen c't-Ausgabe ausführlich gezeigt [1], diesmal stehen die höhere Schule und Stoff für Fortgeschrittene auf dem Programm: etwa der Umgang mit selbst angelegten Klassen und der Einsatz von Webformularen zur Eingabe.

Zwar kann man, wie im ersten Teil gezeigt, einfach Methoden definieren und auf Teile des gerade geöffneten SketchUp-Modells anwenden. Betreibt man das in großem Stil, wird es aber irgendwann eng im Namensraum des SketchUp-API und man muss zu immer umständlicheren Bezeichnern greifen, um sich nicht mit den Namen zu verheddern. Für klarere Gliederung sorgt die Arbeit mit Klassen und Klassenhierarchien – schließlich ist Ruby eine objektorientierte Programmiersprache.

Üblicherweise verpackt man bei Ruby die Methoden in eine Klasse, die wiederum in einem Modul liegt. Solche Module verwendet man in Ruby aus zwei Gründen: Zum einen erlauben sie, mehrere Klassen in einem Namensraum zu bündeln, vergleichbar etwa mit dem Schnüren von Packages in Java. Zum anderen bieten die Module einen Ersatz für die Mehrfachvererbung, die in Ruby nicht möglich ist. Statt eine Klasse von mehreren Oberklassen abzuleiten, kann man Module in Klassen hineinmixen, um deren Methoden in der Klasse zur Verfügung zu haben. Verzagen Sie nicht, falls Sie noch keine Erfahrung mit objektorientierter Programmierung haben und Sie daher mit Begriffen wie Mehrfachvererbung und Oberklasse nichts anfangen können – die folgenden Beispiele zeigen den Umgang mit Modulen in der Praxis. Wie immer finden Sie alle Codebeispiele zum Herunterladen und Ausprobieren über den c't-Link am Artikelende.

### Facettenreich

Wer in SketchUp ein Gebäudemodell mit einer Kuppel bauen will, stellt fest, dass dem Zeichenprogramm ein Werkzeug fehlt, um Kugeln zu zeichnen. Kein Problem, Ruby macht das schon.

Allerdings baut das folgende Beispiel keine echte Kugel, sondern einen beliebig fein facettierten Polyeder als Annäherung. SketchUp lässt nichts anderes zu – aus Performance-Gründen stückelt das 3D-Programm ohnehin jeden Bogen aus geraden Strecken und jede Wölbung aus Polygonen zusammen.

Das Beispielskript ist zunächst in ein Modul und damit in einen eigenen Namensraum gekapselt. Der Kürze halber heißt das Modul hier „Formen". Die Kugel steckt in einer Klasse, die diesem Modul hinzugefügt wird:

```
module Formen
  class Kugel
    def self.dialog
      ...
    end
  end
end
```

Das Kugelwerkzeug soll über ein Icon in einer neuen Leiste namens „Formen" erreichbar sein. Beides legt man ganz ähnlich wie die eigenen Menüeinträge in [1] an. Den Code dafür finden Sie am Ende des kompletten Kugel-Skript-Codes rechts. Die Werkzeugleiste erscheint zunächst als schwebende Palette, lässt sich aber per Drag & Drop zwischen die anderen Leisten am oberen oder linken Rand des Fensters einklinken. Von Hand blendet man die Werkzeugleiste über das Ansicht-Menü und den Eintrag „Funktionspaletten" (auf dem Mac) oder „Symbolleisten" (unter Windows) ein und aus.

Ein Klick auf das Icon ruft die Methode `Kugel.dialog` auf. Sie ist eine Klassenmethode von `Kugel`. Das bedeutet, man kann die Methode auf der Klasse selbst aufrufen, ohne zuvor Instanzen davon erzeugt zu haben, also ohne konkrete Kugel-Objekte. Diese ent-

stehen erst anschließend, indem die Methode dialog über eine weitere Klassenmethode namens new den Konstruktor der Kugel-Klasse aufruft. Die Konstruktor-Methode heißt in Ruby einheitlich initialize. Im Fall der Kugel aus dem Beispiel erwartet sie als Parameter den Radius in Zentimetern sowie die Zahl der Segmente. Je höher die Segmentzahl, desto runder erscheint die Kugel.

## Weltenbau

Der Kugel-Konstruktor überführt die Parameter erst einmal in sogenannte Instanzvariablen, deren Namen in Ruby stets mit @ beginnen. Auf solche Variablen kann man mit allen Methoden zugreifen, die dieselbe Instanz betreffen, sodass man die Parameter bei den folgenden Methodenaufrufen nicht immer wieder neu übergeben muss.

Den Bauplan für die Beispiel-Kugel kann man sich anhand eines Globus veranschaulichen: Der Parameter segmente bestimmt, aus wie vielen horizontal umlaufenden Facettenbändern (@reihen) der Körper konstruiert wird. Beim Mindestwert von 2 entsteht oberhalb und unterhalb des Äquators nur je ein Band, das Ergebnis ist ein Oktaeder. Wählt man hingegen 36 Segmente, erscheint die Kugel in 10-Grad-Teilung. Damit die horizontale und die vertikale Auflösung identisch ausfallen, wird die Segmentzahl für die Zahl der Längenkreise (@spalten) verdoppelt. Die Gesamtzahl der Facetten der Kugelannäherung beträgt dadurch $2 \cdot \text{segmente}^2$. Den Winkel im Bogenmaß, den ein einzelnes Segment abdeckt, berechnet die Zeile schrittweite = 2 * Math::PI / @spalten.

Die Methode punkte_fuer_kugel legt ein zweidimensionales Array an, das die Schnittpunkte zwischen jedem Breiten- und jedem Längengrad speichert. Die verschachtelten Arrays werden aus den Ausdrücken (1…@reihen) und (0…@spalten) erzeugt. Ruby ergänzt diese Bereichsangaben zu Aufzählungen von Startwert bis zum Ende; die Schlusswerte @reihen und @spalten sind dabei in den Aufzählungen nicht enthalten. Die Methode to_a erzeugt aus allen Werten im spezifizierten Bereich jeweils ein Array, auf das dann die Methode map angewandt wird. Diese pickt sich nacheinander jedes Element des Arrays heraus und wendet auf jedes einzeln die folgende Anweisung an – im Beispiel schreibt sie die aktuellen Elemente in die Variablen reihe und spalte, die dann als Parameter für die Methode punkt_fuer verwendet werden.

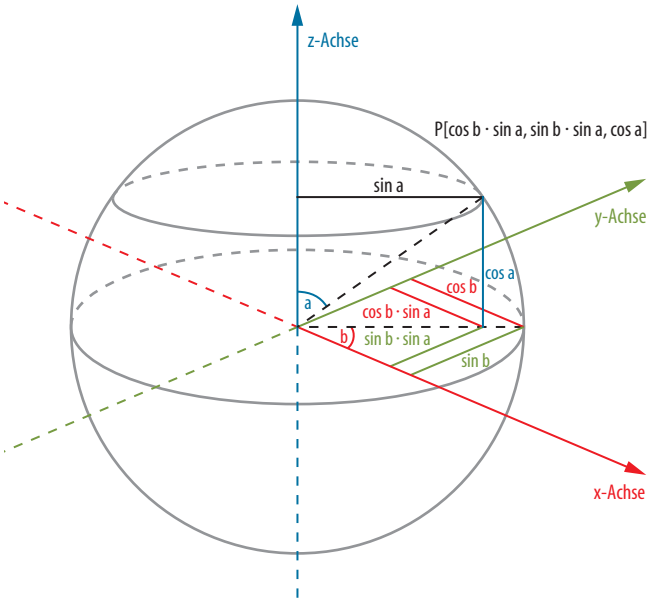
Die Methode punkt_fuer wiederum benutzt einfache trigonometrische Funktionen, um die drei Koordinaten des Punkts in ein weiteres Array zu schreiben. Der Mittelpunkt der Kugel liegt dabei zunächst im Ursprung des Koordinatensystems. Die z-Koordinate ist für alle Punkte auf demselben Breitenkreis (= in derselben Reihe) gleich, sie ergibt sich aus Math.cos(schrittweite * reihe). Die x- und die y-Koordinate für Punkte auf dem Äquator folgen aus dem Cosinus und dem Sinus von schrittweite * spalte. Diese Werte müssen bei Punkten auf allen anderen Breitenkreisen allerdings noch mit deren geringeren Radien verrechnet werden, weshalb man die x- und die y-Koordinate jeweils mit Math.sin(schrittweite * spalte) multipliziert. Die Multiplikation aller drei Koordinaten mit radius bringt die Kugel schließlich auf die gewünschte Größe.

## Schnitz für Schnitz

Sind alle Punkte angelegt, zieht die Methode flaechen_hinzufuegen zwischen jeweils benachbarten Punkten Flächen auf. Sie baut die Kugel ringsherum wie aus einzelnen Apfelschnitzen auf. Die Außenfläche jedes einzelnen Schnitzes wird in drei Stufen erzeugt,

**Das Beispielskript zeichnet regelmäßige Polyeder als Annäherung an die Kugelform, die SketchUp nicht mathematisch genau darstellen kann.**

```
require 'sketchup'
require 'werkzeuge'
module Formen
  class Kugel
    # Methoden aus lib/werkzeuge.rb einfügen
    include Formen::Werkzeuge
    def self.dialog
      # Beschriftung der Dialogfelder und Vorgabewerte
      namen = ['Radius (in cm)', 'Anzahl Segmente']
      werte = [20, 10]
      # Werte werden als Array zurückgegeben
      radius, segmente = UI.inputbox namen, werte, "Kugel"
      # Komponentendefinition erzeugen und Instanz platzieren
      Kugel.new(radius.cm, segmente).komponente_platzieren
    end

    # Kugel-Konstruktor
    def initialize(radius, segmente)
      # Parameter in Instanzvariablen speichern
      @radius = radius
      # Zahl horizontaler Reihen entspricht der Segmentzahl
      @reihen = segmente
      # Zahl vertikaler Spalten entspricht doppelter Segmentzahl
      @spalten = segmente * 2
      # Komponentendefinition erzeugen
      @definition = Sketchup.active_model.definitions.add "Kugel"
      # Punkt, an dem der Körper platziert wird (Kugelmittelpunkt)
      @definition.insertion_point = Geom::Point3d.new(0, 0, -@radius)
      # Punkte erzeugen, in zweidim. Array schreiben, Flächen einfügen
      punkte = punkte_fuer_kugel
      flaechen_hinzufuegen(punkte)
    end

    def komponente_platzieren
      # place_component macht Komponente mit der Maus platzierbar
      modell.place_component @definition
    end

    # Legt für jeden Schnittpunkt zwischen einem Längen- und einem
    # Breitengrad je einen Punkt an; gibt ein zweidimensionales Array zurück
    def punkte_fuer_kugel
      # alle Reihen und Spalten (Breiten- und Längengrade) durchgehen
      # und je einen Punkt erzeugen. Pole werden später separat hinzugefügt.
      (1...@reihen).to_a.map do |reihe|
        (0...@spalten).to_a.map do |spalte|
          punkt_fuer(reihe, spalte)
        end
      end
    end

    # Erzeugt einen dreidimensionalen Punkt an der angegebenen Position
    def punkt_fuer(reihe, spalte)
      schrittweite = 2 * Math::PI / @spalten
      [ @radius * Math.cos(schrittweite * spalte) * Math.sin(schrittweite * reihe),
        @radius * Math.sin(schrittweite * spalte) * Math.sin(schrittweite * reihe),
        @radius * Math.cos(schrittweite * reihe) ]
    end

    def flaechen_hinzufuegen(punkte)
      # Die Punkte in der obersten und der untersten Reihe werden jeweils
      # mit dem Nord- oder Südpol zu einem Dreieck verbunden. Bei allen
      # anderen Reihen entstehen Vierecke.
      @spalten.times do |spalte|
        # oben:
        @definition.entities.add_face([
          [0,0,@radius],
          punkte.first[spalte],
          punkte.first[spalte-1]
        ])
        (punkte.size - 1).times do |reihe|
          # Mitte:
          @definition.entities.add_face([
            punkte[reihe][spalte-1],
            punkte[reihe][spalte],
            punkte[reihe + 1][spalte],
            punkte[reihe + 1][spalte - 1]
          ])
        end
        # unten:
        @definition.entities.add_face([
          punkte.last[spalte-1],
          punkte.last[spalte],
          [0,0, -@radius]
        ])
      end
    end
  end
end
unless file_loaded? File.basename(__FILE__)
  # Toolbar-Icon wird durch UI::Command definiert
  cmd = UI::Command.new("Kugel") do
    Formen::Kugel.dialog
  end
  # zwei Bilder für große und kleine Toolbar-Icons
  cmd.small_icon = File.join(File.dirname(__FILE__),'bilder','kugel_klein.png')
  cmd.large_icon = File.join(File.dirname(__FILE__),'bilder','kugel.png')

  # neue Toolbar für die Icons erzeugen
  toolbar = UI::Toolbar.new "Formen"
  # Icons hinzufügen
  toolbar = toolbar.add_item cmd
  # Toolbar anzeigen
  toolbar.show

end
file_loaded File.basename(__FILE__)
```




**Bei einer Kugel mit Radius eins, deren Zentrum im Ursprung des Koordinatensystems liegt, ergeben sich die Koordinaten eines beliebigen Oberflächenpunkts aus einfachen trigonometrischen Funktionen. Die Winkel a und b resultieren jeweils aus dem aktuellen Zähler für Reihe und Spalte sowie aus der Schrittweite, die dem Winkel entspricht, den jedes Segment abdeckt.**

denn die Methode muss die Pole speziell behandeln. SketchUp beklagt sich nämlich bitterlich, wenn man beim Erzeugen von Flächen einen Punkt zweimal angibt. Deshalb wird zuerst der Nordpol mit den beiden Punkten der obersten Reihe in der aktuellen Spalte und in der vorangegangenen Spalte verbunden, dann werden reihenweise die viereckigen Flächen zwischen der aktuellen Spalte und der vorangegangenen geschlossen. Schließlich verbindet man die beiden Punkte in der untersten Reihe mit dem Südpol.

Hat man erst einmal eine gekapselte Klasse angelegt, spricht nichts dagegen, darin lauter kleine, aber übersichtliche Methoden zu bauen. Dies entspricht den Gepflogenheiten der objektorientierten Entwicklung. Widerstehen Sie vor allem der Versuchung, die komplette Kugel innerhalb der initialize-Methode zu drechseln – das führt zu unnötig langen Routinen.

## Exzentrisch

Die Kugeln stets um den Nullpunkt des Koordinatensystems zu zeichnen, reicht vielleicht zum Testen. Will man das Werkzeug aber während der Arbeit an einem komplizierten Modell produktiv benutzen, ist das unpraktisch.

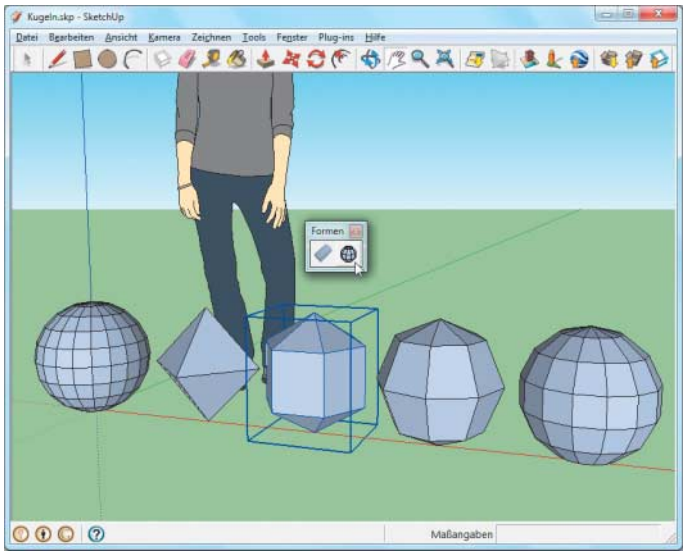
Sogenannte Komponenten fassen in SketchUp einzelne Formen zu Bauteilen zusammen. Es ist nur wenig Code nötig, um die Kugel-Klasse dazu zu bringen, statt einer Gruppe eine Komponente zu erzeugen. Dazu genügen im Konstruktor die beiden Zeilen

```
@definition = Sketchup.active_model.definitions.add "Kugel"
@definition.insertion_point =
    Geom::Point3d.new(0, 0, -@radius)
```

Über definitions.add fügt man dem Modell die Definition einer Komponente hinzu. Genau wie eine Gruppe verfügt sie über ein Array von Entities, eine Sammlung aller zugehörigen Teile. Insofern unterscheidet sich das Erzeugen einer Kugel als Komponente kaum von ihrer Fabrikation als Gruppe.

Ist die Komponenten-Definition vollständig, erzeugt man über modell.place_component eine Instanz des Objekts, die der Benutzer mit der Maus im Raum platziert. SketchUp blendet für solche Komponenten eine einhüllende Box sowie Anfasser ein, über die man sie bequem nachträglich verschiebt und rotiert.

Das Kugel-Skript enthält darüber hinaus ein rudimentäres Beispiel für Methoden, die über ein eingebundenes Modul in die Klasse hineingemixt sind: Die Zeile include Formen::Werkzeuge stellt die Methode gruppe_aus_auswahl aus dem vorangegangenen Artikel auch innerhalb der Kugel-Klasse zur Verfügung.




**Das Icon in der neuen Werkzeugleiste produziert Kugeln und Polyeder wie am Fließband. Von links nach rechts beträgt die Segmentezahl der einzelnen Exemplare zehn, zwei, drei, vier und sieben.**

## Hochgeschraubt

Bisher erzeugten unsere Codebeispiele nur solche Formen, die man mit einiger Handarbeit auch mit der Maus und den serienmäßigen Werkzeugen von SketchUp hinbekommen würde – einen alternativen Weg zum Kuppelbau beschreibt beispielsweise ein Artikel auf heise online (siehe c't-Link). Bei technischen Konstruktionen sind aber oft deutlich komplexere Formen gefragt, Schrauben oder Gewinde etwa.

Prinzipiell ist es gar nicht schwer, ein Gewinde zu konstruieren: Man definiert einen Kernzylinder und berechnet drei Spiralen drumherum. Zwei von ihnen liegen auf dem Zylindermantel und legen den oberen und den unteren Rand des Gewindegangs fest, die dritte markiert den weiter außen liegenden Scheitel des Gangs. Zwischen die drei Spiralen fügt man Flächen ein – fertig.

Soll das Gewinde nicht nur virtuell erzeugt werden, sondern beispielsweise mit einer Maschine wie dem MakerBot (siehe S. 110) dreidimensional gedruckt und anschließend praktisch benutzt werden, muss man Sonderfälle betrachten, etwa beim Abschluss am Ende der Schraube. Außerdem gelten für technische Gewinde besondere Maße, beispielsweise bei den Flankenwinkeln. Insgesamt gibt es bei einem Gewinde sehr viele Kenngrößen, die teilweise voneinander abhängen: So kann bei einem festgelegten Flankenwinkel die Differenz zwischen Außen- und Innenradius nicht beliebig groß sein. Diese Komplexität können die in den bisherigen Beispielen benutzten GUI-Dialoge nicht mehr fassen.

Hier springen sogenannte Webdialoge in die Bresche. Sie erscheinen im Browser und können das gesamte Arsenal an üblichen Webtechniken wie HTML, CSS oder JavaScript verwenden, um dem Nutzer die Konfiguration seines Wunschobjekts so bequem wie möglich zu machen. Es ist sogar denkbar, mit Hilfe von JavaScript und einem <canvas>-Element die gewählten Einstellungen in einer dynamischen Grafik zu visualisieren. Hier verzahnt SketchUp das Ruby-API eng mit dem JavaScript des Webdialogs.

## Browser-Bett

Der Webdialog läuft in einem eingebetteten Browser. Windows greift dabei auf Internet Explorer zurück, Mac OS X auf Safari. Möchte man seine SketchUp-Erweiterung veröffentlichen, sollte man sie mit den aktuellen Versionen der Browser auf beiden Plattformen testen und beim Bau seines Dialogs bei Technologien bleiben, die beide unterstützen.

Ein minimales Beispiel für einen Webdialog erzeugen die Zeilen:

```
web_dialog = UI::WebDialog.new("Titel",
    false, 'hello-world-dialog')
web_dialog.set_html("<html><body><h1>Hallo Welt!
    </h1></body></html>")
web_dialog.show
```

Bei kleinen Dialogen ist es oft sinnvoll, HTML direkt in den Ruby-Code zu schreiben. Alternativ kann man über set_file eine lokale HTML-Datei in den Dialog laden:

```
web_dialog.set_file(File.join(File.dirname(__FILE__),
    'html', 'index.html'))
```

Diese Zeile lädt aus dem Unterverzeichnis „html“ die Datei index.html und benutzt sie als Inhalt für den Dialog.

Die Eingabemaske für das Gewinde-Skript verwendet ein HTML-Formular. Im Web findet man dazu ausführliche Dokumentationen (siehe c’t-Link am Ende des Artikels). Kern des Formulars ist das umschließende <form>-Element. Darin legt das Attribut action das Ziel für das Formular fest. SketchUp adressiert man über eine spezielle URL:

```
<form id="gewinde" action="skp:gewinde_bauen@"
    method="get"> <input type="text"
    name="innenradius" id="innenradius" /> </form>
```

Sendet man dieses Formular ab, versucht der Browser, ein Ruby-Callback aufzurufen. Dessen Gerüst in einem SketchUp-Skript in Ruby folgt der Form:

```
web_dialog.add_action_callback(
    "gewinde_vorgaben_ausfuellen") do |dialog, parameter|
...
end
```

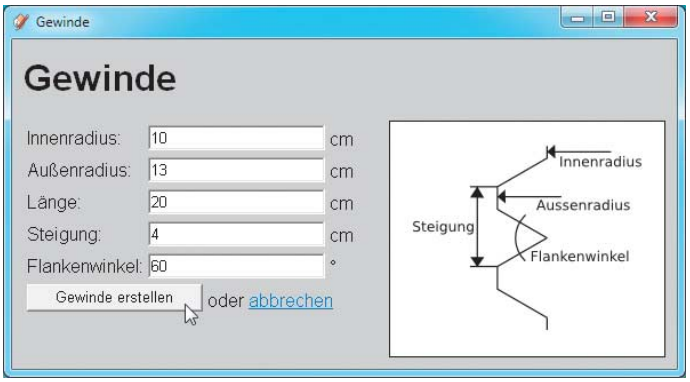
Dadurch kann man innerhalb des oben ausgesparten Codeblocks zwischen add_action_callback und end auf den Webdialog (dialog) und auf eine Zeichenkette namens parameter zugreifen, die alle Daten aus dem Formular enthält. Statt die Benutzereingabe zu Fuß aus der Zeichenkette herauszuklauben, greift man lieber zur Methode get_element_value des WebDialog-Objekts, mit der man direkt an die Werte herankommt, deren Bezeichner im Webdialog definiert wurden. Da die Eingabefelder von HTML-Formularen stets Zeichenketten (Strings) enthalten, muss der Rückgabewert noch mit Hilfe der auf allen Strings definierten to_f-Methode in eine Gleitkommazahl verwandelt werden:

```
innenradius =
    dialog.get_element_value("innenradius").to_f
```

## Sinnfrage

Für zusätzliche Funktionen des Formulars steht JavaScript zur Verfügung. Unser Schraubenbeispiel greift etwa auf eine JavaScript-Bibliothek namens jQuery zurück, die viele Standardaufgaben vereinfacht und vor allem die diversen Unterschiede zwischen Safari und IE geschickt wegkapselt [2].

Im Beispiel-Schraubenformular wird vor dem Abschicken per JavaScript überprüft, ob die Eingaben überhaupt sinnvoll sind. Passt beispielsweise der Flankenwinkel des Gewindes nicht zur Dicke, die sich durch Außenradius minus Innenradius berechnet, käme ein sehr seltsam aussehendes Gebilde heraus. Zudem sollte die Länge mindestens so groß sein wie die Steigung, damit das Gewinde wenigstens eine volle Umdrehung aufweist.




**Web-Dialoge als Eingabemasken für Ruby-Skripte lassen sich flexibel layouten.**

Der folgende Codeschnipsel zeigt eine solche Prüfung. Per jQuery-Bibliothek wird dabei ein sogenannter Event-Handler in den Versendevorgang des HTML-Formulars eingeklinkt: Gibt er false zurück, geht das Formular erst gar nicht auf die Reise.

```
$(function() {
  $('#gewinde').submit(function(e) {
    var innenradius = parseFloat($('#innenradius').val());
    var aussenradius = parseFloat($('#aussenradius').val());
    var dicke = aussenradius - innenradius;
    var minimalsteigung = Math.tan(angle/2) * dicke;
    if (minimalsteigung > (steigung/ 2)) {
      alert("Die Werte ergeben kein wohlgeformtes
          Gewinde!");
      return false;}
    return true;});
});
```

Die umhüllende Funktion sorgt dafür, dass der Eventhandler erst installiert wird, wenn der Inhalt des Webdialogs komplett vom integrierten Browser geladen ist. Andernfalls könnte es passieren, dass jQuery das Formular nicht findet. Eine solche Kapselung in die $()-Funktion ist immer sinnvoll, wenn man Eventhandler installieren möchte.

Eine weiter ausgefeilte Validierung finden Sie in unserem Beispielcodepaket im Unterverzeichnis html/javascript in der Datei gewinde.js – sie prüft beispielsweise zusätzlich, ob der Innenradius kleiner ist als der Außenradius.

## An die Tasten

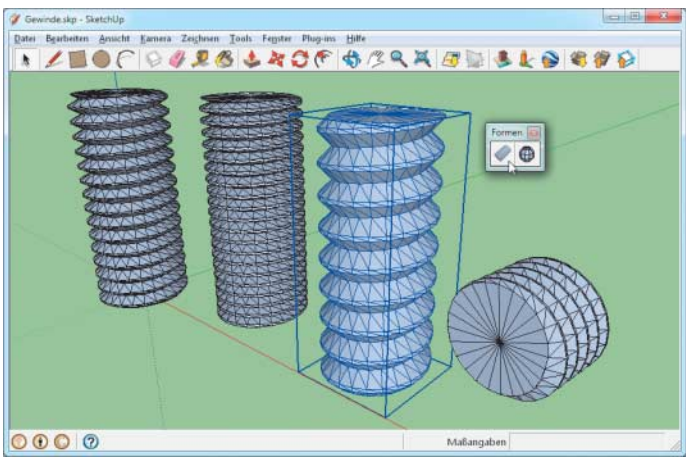
Den eigentlichen Algorithmus zum Schraubendrehen erläutern ausführliche Kommentare im Code, der hier aus Platzgründen nicht abgedruckt wird. Das Skript lässt sich in vielerlei Richtungen weiter treiben – beispielsweise könnte das Formular einen zusätzlichen Parameter für die Segmentzahl einfordern, sodass Gewinde aus mehr Facetten pro Umdrehung zusammengesetzt würden als die im Beispiel hart codierte 24-Teilung. Auch die Vorabprüfbedingungen sind noch keineswegs lückenlos. Und wie wäre es mit einer Variante, die Normgewinde wie M6 oder M8 erzeugt? Ob Sie unsere Beispiele als Grundlage für eigene Entwicklungen benutzen, unsere Werkzeuge einfach nur für den Bau von 3D-Modellen praktisch einsetzen oder mit dem leeren Editorfenster von Grund auf eigene Skripte für SketchUp entwickeln – wir freuen uns über jedes Ruby-Skript und jedes 3D-Objekt, das Sie uns mailen. (pek)

### Literatur

[1] Jan Krutisch, Konstruieren mit Code, Eigene Werkzeuge für SketchUp programmieren, c’t 2/11, S. 158

[2] Daniel Koch, Rahmenarbeiter, JavaScript-Frameworks erleichtern Web-Projekte, c’t 8/10, S. 154

***www.ct.de/1103180*** ct




**Das Beispielskript erzeugt Gewindestangen mit frei wählbaren Parametern für Länge, Durchmesser, Steigung und Flankenwinkel. Vorher überprüft der Eingabedialog, ob die Werte zueinander passen.**