Anhang 1: Entwicklung eines Dynamischen Geometrieprogramms

In diesem Anhang wird vordergründig die Entwicklung eines dynamischen Geometriesystems beschrieben, hintergründig geht es aber darum, Schüler in eine unübersichtliche Situation zu führen und sie so erleben zu lassen, wie OO hilft, solche Probleme zu bewältigen. Dabei kommt ein selbstgebautes Klassensystem zum Einsatz wie in Weg 2 des Aufsatzes beschrieben.

Voraussetzungen

Als Werkzeug wird die Programmiersprache *Python 2.x, x>4* verwendet. Ferner kommt eine kleine Zusatzbibliothek SWGUI zum Einsatz.

Zielformulierung

Es soll ein dynamisches Geometrieprogramm erstellt werden, so ähnlich wie *Geogebra* oder *Euklid*, die vermutlich aus dem Mathematikunterricht bekannt sind. Als Objekte soll es geben: Punkte, Geraden und Kreise. Konstruktionswerkzeuge sollten sein: Mittelpunkt, Schnittpunkt, Lot usw.

1 Die grafische Benutzeroberfläche

Auf der Oberfläche sind die geforderten Schaltflächen (Knöpfe) sowie die Zeichenfläche zu platzieren. Die Ereignisbehandlung ist soweit zu realisieren, dass schon zwischen den verschiedenen Modi des Programms (z.B. Punkte setzen, Geraden erzeugen, …) unterschieden werden kann. Außerdem sollte schon zwischen Bildschirmkoordinaten und mathematischen Koordinate umgerechnet werden können. Eine Lösung ist in der Datei *DGS1.py* enthalten. Mit den 100 Zeilen dieser Datei hat man immerhin schon ein paar Ergebnisse erzielt: Es lassen sich Punkte erzeugen und man kann Klicken und Ziehen – wenn auch mit wenig Effekt. Wichtig ist, dass hier an mehreren Stellen Funktionen als Parameter vorkommen: Bei der *map*-Funktion und bei allen Befehlen, die Ereignisbehandler setzen.

2 Geometrische Objekte

Wie sollten die geometrischen Objekte beschrieben werden, welche Informationen sind notwendig? Bleiben wir zunächst bei den Punkten. Jeder Punkt sollte einen Namen haben, der auch mit angezeigt wird. Außerdem müssen die mathematischen Koordinaten gespeichert werden, nicht nur die Bildschirmkoordinaten. Es bietet sich an, diese verschiedenen Dinge in eine Liste oder einem Dictionary zu speichern. Letzteres hat den Vorteil, dass man sich nicht merken muss, wo in der Liste welche Information steht. Außerdem speichern wir alle Objekte in einer Liste, das ist sicher nützlich, wenn man beispielsweise die ganze Zeichnung speichern will. Damit könnte eine ganz einfache Punkterzeugung so aussehen:

```
def erzeugePunkt(co): # Co ist Liste von Weltkoordinaten
  global zeichenflaeche,objekte
  [xs,ys]=welt2schirm(co)
  P = {"x":co[0], "y":co[1], "name":punktName()}
  # P ist Dictionary mit allen Informationen
  GUIpunkt=zeichenflaeche.createCircle(xs,ys,5)
  GUIpunkt.setColor("red")
  P["gui"]=GUIpunkt
  P["label"]=zeichenflaeche.createText(xs+15,ys,P["name"])
  objekte.append(P)
```

In einem DGS können die Punkte mit der Maus gezogen werden. Wenn man aus zwei Punkten A und B den Mittelpunkt M konstruiert hat, und A gezogen wird, dann muss M mitbewegt werden. Eine eingebürgerte Sprechweise nennt A und B die *Eltern* von M und M das *Kind* von A und B. Diese Beziehung lässt sich auch schön mit einem gerichteten Graphen darstellen. Wenn also die Eltern „umziehen“, müssen die Kinder mit umziehen. Damit man weiß, wer das ist, muss die Erzeugung entsprechend ergänzt werden. Außerdem muss man speichern, ob ein Punkt frei ist, oder von einem bestimmten Typ:

```
P["typ"]="Punkt"
P["untertyp"]=""
P["frei"]=True # Ist der Punkt frei ziehbar?
P["kinder"]=[]
P["eltern"]=[]
```

Die Erzeugung eines Mittelpunktes aus zwei solchen Punkt-Objekten sieht so aus:

```
def erzeugeMittelpunkt(ops):
  global zeichenflaeche,objekte
  [A,B]=ops
  co=[ (A["x"]+B["x"])/2.0, (A["y"]+B["y"])/2.0 ]
  [xs,ys]=welt2schirm(co)
  P={"x":co[0], "y":co[1], "name":punktName()}
  # P ist Dictionary mit allen Informationen
  GUIpunkt=zeichenflaeche.createCircle(xs,ys,5)
  GUIpunkt.setColor("blue")
  P["gui"]=GUIpunkt
  P["label"]=zeichenflaeche.createText(xs+15,ys,P["name"])
  P["typ"]="Punkt"
  P["untertyp"]="Mittelpunkt"
  P["frei"]=False # Ist der Punkt frei ziehbar?
  P["kinder"]=[]
  P["eltern"]=ops
  objekte.append(P)
  A["kinder"].append(P)
  B["kinder"].append(P)
```

Wenn mit der Maus ein Objekt gezogen wird, ruft man moveTo für dieses Objekt auf. Es platziert es neu und informiert gleich alle Kinder, dass sie sich auch „updaten“ müssen:

```
def moveTo(obj,co): # Bewegt das Objekt obj nach co
  if obj["typ"] <> "Punkt": return # Nur Punkte sind ziehbar
  if obj["frei"]:
    [obj["x"],obj["y"]]=co
    [sx,sy]=welt2schirm(co)
    obj["gui"].setCenter(sx,sy)
    obj["label"].setCoords([sx+15,sy])
    for kind in obj["kinder"]: update(kind)

def update(obj): # Eltern haben sich bewegt, jetzt aktualisieren
  if obj["typ"]=="Punkt" and obj["untertyp"]=="Mittelpunkt":
  [A,B]=obj["eltern"]
  co=[ (A["x"]+B["x"])/2.0, (A["y"]+B["y"])/2.0 ]
  [obj["x"],obj["y"]]=co
  [sx,sy]=welt2schirm(co)
  obj["gui"].setCenter(sx,sy)
  obj["label"].setCoords([sx+15,sy])
```

```
  for kind in obj["kinder"]: update(kind)
```

Zusammen mit dem Programmtext für die Ereignisverarbeitung macht das ein knapp 200-zeiliges Programm, das immerhin Mittelpunkte richtig aktualisiert: *Listing DGS2.py*. Man könnte direkt darin weiterprogrammieren und die andere Objekte analog erzeugen, und es ist eine Frage, ob man die Schüler das auch tun lassen sollte. Es wird funktionieren, aber man kann folgende Feststellungen machen (die sich bei genauer Beobachtung auch schon jetzt im Programmtext zeigen):
- Es gibt viele Doppelungen.
- Wenn ein neuer Objekttyp eingeführt wird, muss man den ganzen Quelltext durchpflügen, weil das beispielsweise das Wissen über Punkte weit verstreut ist. Es wäre praktischer, das zusammenzuhalten.
- Die vielen Typ- und Untertyp-Abfragen sind umständlich.

Man kann mit konventioneller Programmierung diese Probleme etwas entschärfen, aber nicht richtig lösen. Natürlich kann man das Klassensystem von Python benutzen, aber wir haben ja schon angefangen, die Objekte als Dictionaries zu realisieren, und so kann man auch weitermachen.

3 Selbstgebaute Objekte

Jede Klasse geometrischer Objekte muss aktualisierbar sein. Also wird eine entsprechende Funktion im Konstruktor der Klasse gespeichert, so wie das im Hauptartikel an einem etwas vereinfachten Beispiel erläutert wurde. Sicher ist es an dieser Stelle nötig, dass die Lehrkraft leitend wirkt, denn man muss einige Entscheidungen vorausschauend treffen, um anschließend beispielsweise die Aktualisierungsmethode in einem einheitlichen Format aufrufen zu können. Das Ergebnis ist in *DGS3.py* zu besichtigen. Diese Fassung ist funktional äquivalent zu *DGS2.py*, aber wesentlich übersichtlicher.

4 Alles nachrüsten, was fehlt!

Bezüglich der Programmlogik passiert jetzt eigentlich nichts mehr Spannendes. Das Nachrüsten weiterer geometrischer Konstruktionswerkzeuge und Objektarten kann den Schülern also weitgehend überlassen werden. Allerdings gibt es an einigen Stellen haarige Mathematik zu bewältigen: Die Schnittpunkte zweier Kreise oder eines Kreises und einer Geraden sind mit Mathematik der Sekundarstufe I berechenbar. Damit man die Formeln in das Programm schreiben kann, muss man die Gleichung aber komplett symbolisch lösen, d.h. die Koordinaten der Kreismittelpunkte und die Radien müssen als Parameter im Gleichungssystem auftreten. Das führt zu recht langen Formeln. Ein Weg ist, diese mit einem Computeralgebra-System auszurechnen. Viele dieser Systeme können Terme in Fortran oder C-Konvention ausgeben, von da nach Python ist es nicht sehr weit. Die Beispielslösung in der Datei *DGS4.py* geht einen anderen Weg: Die Gleichungen werden nummerisch gelöst. Auch das ist erklärungsbedürftig. Ob man hier Zeit investieren will, ist eine Frage der Ziele. Man kann natürlich auch auf Schnitte nichtlinearer Objekte verzichten und lieber noch Strecke, Polygone usw. als Objekte einfügen.