Programmieren mit dem XO-Laptop

von Joachim Wedekind und Christian Kohls

Der XO-Laptop ist mit einer Palette von Werkzeugen ausgestattet, die das Bearbeiten multimedialer Komponenten erlauben (Texte, Bilder, Videos), den Zugang zum Internet eröffnen und kollaboratives Arbeiten unterstützen. Ebenfalls enthalten sind mehrere Entwicklungsumgebungen zum Programmieren. Da die Adressaten primär Grundschüler sind, ist es nicht verwunderlich, dass dabei Anleihen bei der Programmiersprache LOGO genommen wurden. Mit *Turtle Art*, SCRATCH und *Etoys* stehen Umgebungen zur Verfügung, die dies in unterschiedlichem Maße mit Formen der visuellen Programmierung anbieten.

Mit dem XO-Laptop erhalten die Schülerinnen und Schüler auch Zugang zu Entwicklungsumgebungen, mit denen sie Programme entwickeln können. In der Tradition der Computersprache LOGO (siehe Kasten „Die Ursprünge“, nächste Seite) sollen diese fast keine Einstiegshürde bieten und damit erlauben, kreativ, entdeckend und intuitiv Problemlösungen zu erarbeiten. Die angebotenen Werkzeuge wollen deshalb „Ideenverarbeitungsprogramme“ sein, also zugleich Sprache, Werkzeug und Medienentwicklungsumgebung (Allen-Conn/Rose, 2003). Im Folgenden werden einige dieser Werkzeuge vorgestellt.

Visuelle Programmierung

Die Werkzeugpalette des XO-Laptops unter dem Betriebssystem SUGAR setzt stark auf visuelle Programmierungselemente (vgl. Freudenberg, 2009, S. 40 ff., in diesem Heft). Das findet sich wieder in der Schildkrötengrafik (*Turtle Art*), dem Zeichenprogramm *Paint* und dem Musikprogramm *TamTam*.

Eine Programmiersprache ist ein Notationssystem zur Beschreibung von Berechnungen in von Maschinen und Menschen lesbarer Form (Louden, 1994). *Visuelle Programmiersprachen* zeichnen sich dadurch aus, dass zur Beschreibung des Algorithmus visuelle Komponenten mit einbezogen oder ausschließlich verwendet werden (Schiffer, 1998). Damit ist nicht die pragmatische Gestaltung des Programmtextes – z. B. durch Einrückungen oder Syntax-Hervorhebung – gemeint, sondern die Beschreibung der Rechenschritte durch visuelle Elemente, die zur Syntax gehören. Somit legen grafische (Farben, Piktogramme), geometrische (Form, Größe, Seitenverhältnisse) und topologische (Verbindungen, Überlagerungen, Berührungen) Eigenschaften die Semantik fest.

Zu unterscheiden ist ferner zwischen der Programmierung visueller Ausgaben, der visuellen Gestaltung von Benutzungsschnittstellen und der visuellen Festlegung des Programmtextes. LOGO war zwar seit jeher mit einer Schildkrötengrafik ausgestattet, mit der eine Programmierung grafischer Ausgaben möglich ist, doch die Anweisungsfolgen wurden weiterhin textlich (verbal und nicht visuell) festgelegt. Die grafische Gestaltung von Benutzungsoberflächen dagegen ist heute in den meisten integrierten Entwicklungsumgebungen mithilfe von Formular-Editoren möglich.

Als *visuelles Programmieren* wird die Verwendung grafischer Darstellungen in der Softwareentwicklung bezeichnet. Oder genauer (Schiffer, 1996): Eine *visuelle Sprache* ist eine formale Sprache mit visueller Syntax oder visueller Semantik und dynamischer oder statischer Zeichengebung. Kennzeichnender Bestandteil dafür ist eine grafische Notation der Grundsymbole, mit denen computersprachliche Konstrukte repräsentiert werden. Die Hoffnung beim Einsatz visueller Techniken ist, dass insbesondere Programmierlaien der Zugang zur Softwareentwicklung eröffnet wird. Als Vorteile der visuellen Programmierung werden genannt: Der anschauliche Realitätsbezug, großes Motivations- und Lernpotenzial, eine Abschwächung syntaktischer Strukturen und die Betonung semantischer Zusammenhänge. Alles das könnte natürlich für die Nutzung in Lehr-/Lernkontexten sprechen. Mögliche Nachteile dürfen aber nicht verschwiegen werden, wie: fehlende Standards für die Repräsentationen, geringe Darstellungsdichte, eingeschränkte Änderungsmöglichkeiten, schwierige Formalisierbarkeit und beschränkte Abstraktionsmöglichkeiten. Dies alles sind Aspekte, die insbesondere beim Verlassen des absoluten Anfängerniveaus relevant werden können.

Andererseits haben entsprechende Werkzeuge in den Ingenieurswissenschaften Verbreitung gefunden. Visuelle Programmieransätze in Schule und Hochschu-

Die Ursprünge

Es ist kein Zufall, dass mit dem XO-Laptop Werkzeuge in der LOGO-Tradition angeboten werden. Im Beirat des OLPC-Projekts und damit einflussgebend auf Konzeption und Umsetzung sind unter anderen Seymour Papert und Alan Kay. Papert, Mathematiker, Computer- und Erziehungswissenschaftler, hatte Mitte der Sechzigerjahre die Programmiersprache LOGO entwickelt, um damit Kindern einen Zugang zum Programmieren zu eröffnen und den Computer als Werkzeug zum Lösen relevanter Probleme in unterschiedlichen Fächern einzuführen. Alain Kay, gelernter Biologe und Informatiker, traf 1968 Papert und lernte in dessen Labor am MIT diesen Ansatz kennen.

http://www.cc.gatech.edu/

Foto: catypar

http://www.flickr.com/photos/catypar/3217275719/

Alan Kay (links) und andere schufen SMALLTALK, Brian Silverman (rechts) schuf *Turtle Art*.

Das war ein Schlüsselerlebnis für ihn und beeinflusste entscheidend seine Entwicklungsarbeit an grafischen Benutzungsschnittstellen (damals in der Forschungsabteilung Xerox Parc). Er und sein Team mussten dafür erst geeignete Programmierwerkzeuge entwickeln. Daraus entstand die Sprache SMALLTALK. Sowohl die Interface-Entwürfe als auch die Entwicklungsumgebung selbst erprobten Kay und sein Team immer wieder mit Schülergruppen, weil für sie Kinder die „ideale Nutzergruppe" darstellten, wenn sie Werkzeuge zum Lernen und Arbeiten entwickeln wollten.

Als zentrale Komponente von LOGO wurde ab 1970 die Turtle-Grafik (Schildkrötengrafik) eingeführt, durch die mit einfachen Grundbefehlen ansprechende und komplexe Grafiken erzeugt werden können. Jedenfalls, als zu Beginn der Achtzigerjahre noch um die „richtige" Programmiersprache für den schulischen Informatikunterricht gestritten wurde, gehörte LOGO zu den „schulspezifischen" Sprachen, die in Modellversuchen erprobt wurden. Es stand damit in Konkurrenz zu ELAN, PASCAL-E, BASIC oder COMAL-80 (vgl. das Themenheft „Programmiersprachen", LOG IN, 3. Jg. (1983), Heft 3). Für LOGO sprach, dass es als interaktive, listenverarbeitende, prozedurale Programmiersprache durch die lernpsychologisch motivierten Eigenschaften didaktische Grundideen tragen kann (vgl. Tauber, 1983, S. 37):

- ▷ die Unterstützung von entdeckendem, spielerischen Lernen,
- ▷ das Formulieren und Ausprobieren von Vermutungen,
- ▷ das Lernen aus Fehlern und deren Korrektur,
- ▷ die Zerlegung von Problemen in Teilprobleme und deren Lösung mithilfe von Prozeduren,
- ▷ das Baukastenprinzip, d.h. die Erweiterung der Sprache durch eigene problemspezifische Prozeduren,
- ▷ die Erarbeitung von programmiersprachlichen Konzepten mit der Schildkrötengrafik,
- ▷ das einfache interaktive Arbeiten mit sofortiger grafischer Rückkopplung.

le finden sich ebenfalls vielfach und in unterschiedlichen Bereichen. Das bekannteste Beispiel sind Modellbildungssysteme, mit denen dynamische Systeme dargestellt und simuliert werden können. Durch die interaktive Grafik dieser Systeme wird neben der Veranschaulichung der abstrakten Zusammenhänge (ikonische Komponente) und der Hinführung zu formalen Darstellungen (symbolische Komponente) in mathematischen Gleichungen bzw. formallogischen Beziehungen eine hohe Eigenaktivität der Lernenden erreicht (enaktive Komponente), weil mit ihr die vollständige Modell-Implementation, Simulation und Programmsteuerung gewonnen wird (siehe Bild 1).

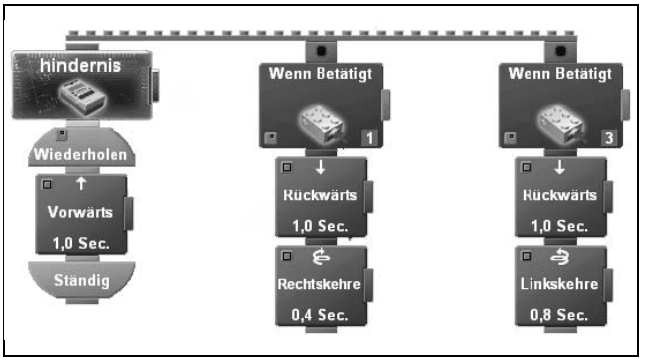
Die grafische Umsetzung komplexer Algorithmen stößt allerdings ihrerseits an eine Grenze, die „Deutsch-Limit" genannt wird (nach dem Informatiker L. Peter Deutsch): Mehr als 50 visuelle Elemente auf dem Bildschirm sind schwer darstellbar und zu verarbeiten.

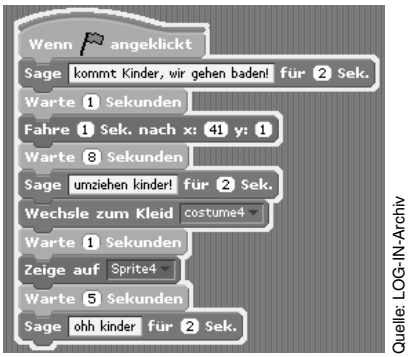
Der Vorteil beim Zusammenstellen des Programms per „Drag & Drop" liegt zum einen in der Vermeidung syntaktischer Fehler. So lassen sich falsch geschriebene oder nicht existierende Anweisungen und Operationen gar nicht erst eingeben. Die richtige Reihenfolge der Syntaxkomponenten wird durch die Form der Blöcke und deren Verbindungspunkte implizit vorgegeben. Die unterschiedlichen Farben und Formen für Anweisungen, numerische Eingaben, Kontrollflüsse und Ausgaben fördert zusätzlich das Verständnis. Diese speziellen typografischen Auszeichnungen haben zwar keinen

Bild 1:
Beispiele visueller Programmierung.

links: Programmierung zur Hindernisvermeidung eines LEGO-MindStorms-Roboter, zusammengestellt in RIS (***R**obotics **I**nvention **S**ystem*).

rechts: Programm eines Kurzfilms, zusammengestellt in SCRATCH.





Quelle: LOG-IN-Archiv

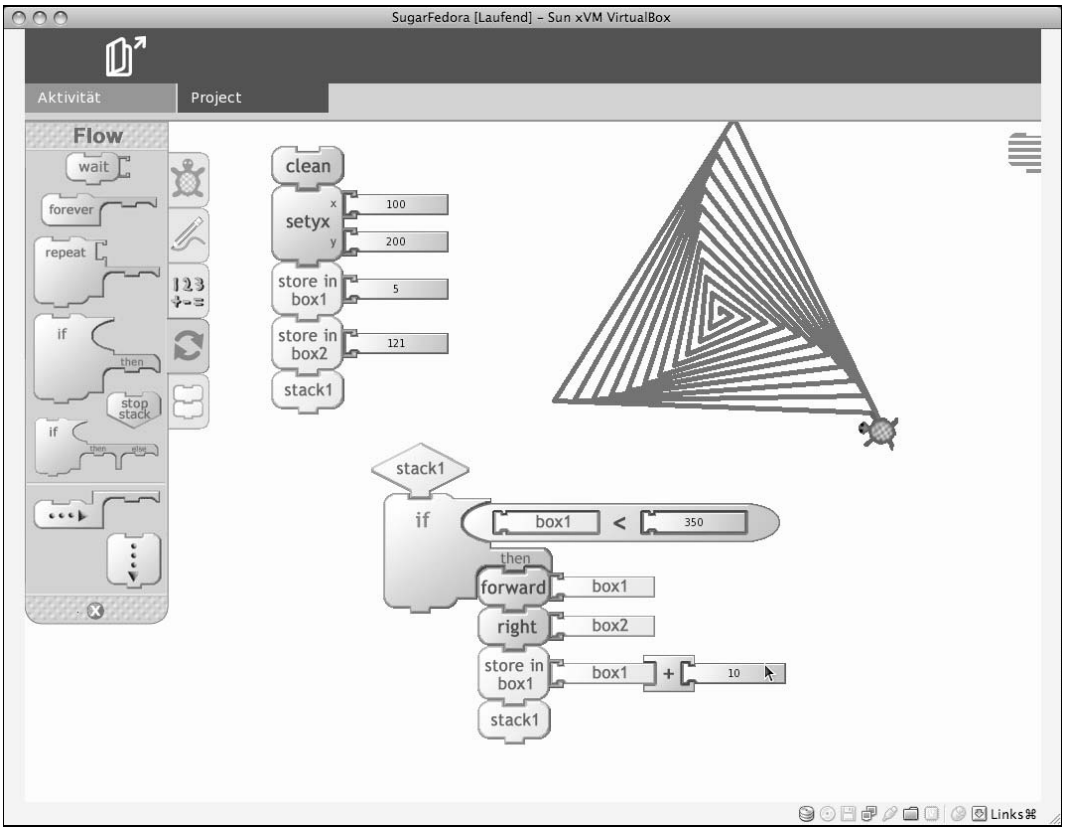


Bild 2: Programmierung einer Spirale in *Turtle Art*, hier umgesetzt mit endständiger Rekursion (stack1).

Einfluss auf Syntax und Semantik (d.h. wenn alle Puzzleteile die gleiche Farbe und Form hätten, würde sich der Algorithmus nicht ändern), sind aber wichtig für die Pragmatik, d.h. die Wirkung und Beziehung der Zeichen zu den Schülerinnen und Schülern. Sie können lernförderlich sein, da sie die Bedeutung der einzelnen Symbole hervorheben, ohne die Semantik zu verändern. So ist z.B. die parallele Darstellung alternativer Anweisungsfolgen wie bei if-then-else beim Erlernen der Programmlogik hilfreich. Da sich die Ablaufsteuerung im Prinzip nicht von der textlichen Programmierung imperativer Sprachen unterscheidet, wird der Wechsel zur textlichen Programmierung (für umfangreichere Programme) erleichtert. Das zur Entwicklung komplexer Programme erforderliche logische Verständnis wird mit den Puzzle-Diagrammen zwar gefördert, ist jedoch weiterhin unentbehrlich.

Turtle Art

Die Schildkrötengrafik ist nur eine (wenn auch die bekannteste) Untermenge der Sprache LOGO, die in völlig verschiedenen Implementierungen immer wieder aufgegriffen und in unterrichtlichen Kontexten eingesetzt wurde. Der überschaubare Befehlssatz, das interaktive Arbeiten und die ansprechenden grafischen Ergebnisse waren Gründe für die bis heute vielfältigen Implementierungen. Gleich mehrere Projekte zu anfassbaren Benutzungsschnittstellen (*tangible programming*) haben ebenfalls auf diesen Ansatz zurückgegriffen. Mit *Crickets* (Rusk u.a., 2008) und *Tern* (Horn/Jacob, 2007) wurden physische Objekte entwickelt, die wie Puzzleteile zu Programmstrukturen zusammengesetzt und mit denen dann entweder Roboter (kybernetische Schildkröten) oder Bildschirmelemente gesteuert werden können. Dem gleichen Ansatz folgen übrigens Zuckerman/Resnick (2003) mit *System-Blocks* bei der Einführung in die Systemdynamik.

Turtle Art ist eine von Brian Silverman (siehe Kasten, vorige Seite), einem LOGO-Pionier, entwickelte Aktivität für SUGAR, die nun diesem Prinzip folgt. Gegenüber der Programmiersprache LOGO bestehen gravierende Einschränkungen (z.B. gibt es nur globale Variablen, keine Listen), da sie sich – wie der Name schon sagt – auf die Schildkrötengrafik beschränkt.

In *Turtle Art* stehen folgende Befehlsarten zur Verfügung:

- ▷ Bewegung der Turtle (wie forward, back, left, right, setxy, heading, arc)
- ▷ Stifteigenschaften (wie pen up, pen down, set pen pensize, set color, set shade, fill screen)
- ▷ Kommandos zur Ablaufsteuerung (wie wait, forever, repeat, if–then–else)
- ▷ Rechenoperationen (Addition, Subtraktion, Multiplikation, Division usw.).

Über (maximal zwei) Stacks können Unterroutinen definiert werden. Das Arbeiten mit *Turtle Art* kann an einem einfachen Beispiel einer rekursiven Grafik erläutert werden (siehe Bild 2).

Umfangreichere Ausformulierungen sind hiermit also kaum möglich, denn dann macht sich sofort die Platzbeschränkung bemerkbar. Insgesamt ist die begrenzte Syntax von *Turtle Art* beim Einsatzspektrum als limitierender Faktor zu beachten. Übrigens ist *Turtle Art* nur auf dem XO-Laptop bzw. unter SUGAR lauffähig.

SCRATCH

SCRATCH, die zweite Entwicklungsumgebung, ist eine visuelle Programmiersprache, die vor allem die Erstellung von Spielen und Multimedia-Anwendungen ermöglicht. Auf einer Bühne (fest vorgegebene 480×360 Bildpunkte) werden beliebig viele Objekte (sogenannte Sprites) gesetzt und über zugeordnete Code-Module gesteuert (vgl. Romeike, 2007). SCRATCH bietet einen deutlich umfangreicheren Befehlssatz als *Turtle Art* (bedauerlicherweise ist dabei kein direkter Übergang möglich, da einige gemeinsame Sprachelemente unterschiedlich repräsentiert werden):

- Bewegung der Sprites (wie Gehe … vorwärts, Drehe um, Zeige auf usw.)
- Aussehen (wie Nächstes Kleid, Ändere Grösse, Verstecken, Erscheinen usw.)
- Malstift (wie Stift runter, Stift hoch, wähle Stiftfarbe usw.)
- Steuerung (wie Wenn angeklickt, Warte, Wiederhole x-Mal, Fortlaufend, Wenn usw.)
- Fühler (Maus-Position, Farbe x wird berührt?, Entfernung von usw.)
- Rechenoperationen (Addition, Subtraktion, Multiplikation, Division usw.)

In SCRATCH können mehrere Sprites gleichzeitig angesprochen werden. Zwischen den Objekten können Nachrichten ausgetauscht werden. Jedem Objekt können eigene Code-Module zugeordnet werden, d. h. sie können autonom agieren und auf Ereignisse bzw. Nachrichten reagieren. Da die Objekteigenschaften und sogar der Programmcode während des Programmablaufs verändert werden kann, wird ein spielerisches und experimentelles Arbeiten geradezu herausgefordert (siehe Bild 3).

Unterroutinen – und damit auch rekursive Aufrufe – sind in SCRATCH nicht möglich: eine Einschränkung gegenüber *Turtle Art*. SCRATCH läuft dafür auf allen Plattformen (Windows, Mac, Linux). Es gibt eine sehr aktive Internetgemeinschaft (http://scratch.mit.edu/), die eine Fülle von Beispielen zusammengetragen hat (derzeit über 300 000 Projekte von über 45 000 registrierten Mitgliedern).

Etoys

Einen Schritt weiter geht das ebenfalls auf dem OLPC-Laptop verfügbare *Etoys*. Es setzt auf der SMALLTALK-Umgebung SQUEAK auf und bietet somit einen konsequent objektorientierten Programmieransatz. Objekte können selbst gezeichnet und mit Variablen und Skripten zum Leben erweckt werden (vgl. Freudenberg/Hancl/Mietzsch 2007). Da sich Objekte mit einfachen Anweisungen über die Spielwiese bewegen lassen und dabei Zeichnungen anfertigen können, sind auch hier Schildkrötengrafiken möglich. Die Skripte werden ebenfalls visuell per „Drag & Drop“ zusammengestellt, allerdings als reine Flussdiagramme, deren einzelne Anweisungs- und Ausdrucksblöcke allesamt gleich aussehen und somit die wirkungsvolle Pragmatik der Puzzle-Diagramme von *Turtle Art* bzw. SCRATCH nicht besitzen. Zudem werden syntaktisch unsinnige oder unvollständige Anordnungen nicht sofort erkenntlich: sicherlich ein Nachteil gegenüber *Turtle Art* und SCRATCH. Auf der anderen Seite eröffnet *Etoys* deutlich mehr Möglichkeiten (siehe Bild 4, nächste Seite).

In *Etoys* kann man etwa durch eine einfache Anweisung festlegen, dass sich ein Objekt in Richtung eines

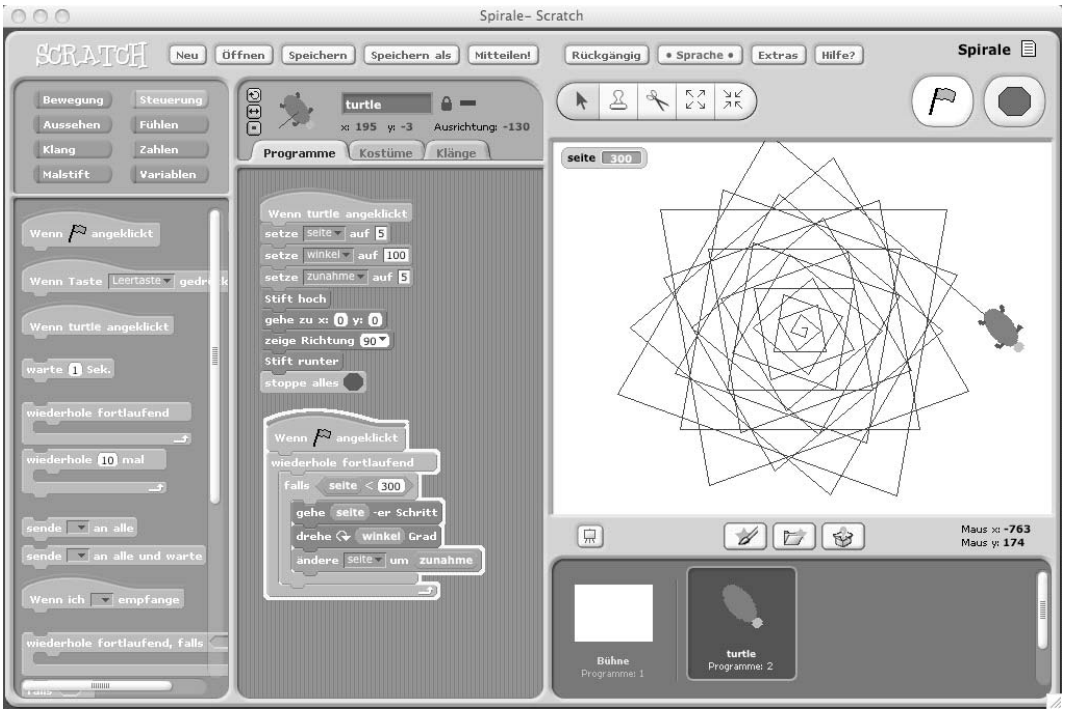


Bild 3: Programmierung einer Spirale in SCRATCH, hier umgesetzt mit Wiederholungsschleife.

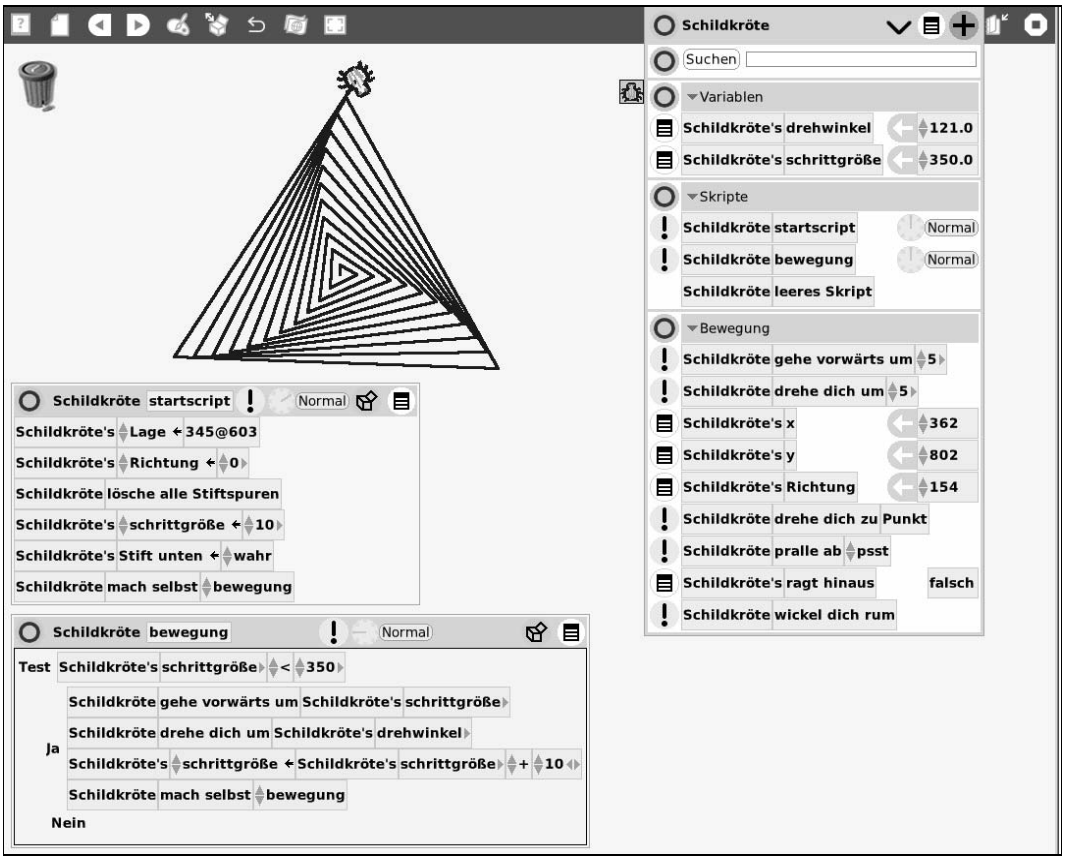


Bild 4: Programmierung einer Spirale in *Etoys*, hier umgesetzt mit Rekursion.

anderen Objekts drehen soll. In Verknüpfung mit einer Test-Anweisung kann dies geschehen, wenn sich ein Objekt zu weit von einem anderen entfernt hat, an die Grenzen der Spielwiese gerät oder auf eine bestimmte Farbe „tritt“. In einem weiteren, parallel ausgeführten Skript lässt sich dann festlegen wie sich das Objekt normal bewegen oder auf die Kollision mit anderen Objekten reagieren soll. Durch die Interaktion mit der Spielwiese kann über die grafische Gestaltung der Umgebung das Verhalten der Objekte visuell definiert werden. Die Zeichnungen sind also mit Bedeutung belegt. Dies gilt auch für die Objekte selbst, die stets visuell repräsentiert sind. Bestimmte Farbpunkte der Objektzeichnung können bestimmen, an welcher Stelle die Umgebung auf der Spielwiese analysiert wird. Eine Schildkröte in *Etoys* folgt also nicht nur festgelegten Anweisungen zur Ablaufsteuerung und bestimmten Bewegungsmechanismen, sondern analysiert auch die Umgebung und ist somit sehr gut für einfache Modellbildungen geeignet.

Zwar sind physikalische Simulationen für den Grundschulbereich sicher zu umfangreich, doch einfache Simulationen, z. B. eines virtuellen Haustiers, das gepflegt werden muss, lassen sich recht einfach realisieren. Dabei werden bereits einige Konzepte der Objektorientierung vermittelt, insbesondere der strukturelle Zusammenhang von Daten und Verhalten. Das Festlegen mehrerer Skripte, die parallel ablaufen und auch auf Ereignisse reagieren können, erlaubt zudem eine intuitivere Definition von Verhaltensregeln, als dies bei (rein) imperativer Programmierung der Fall wäre. So formulieren nach einer Studie von Pane, Myers und Ratanamahatana (2001) Nicht-Programmierer die Regeln von Objekten eher natürlichsprachig wie bei der Schildkrötengrafik (z. B. „Objekt bewegt sich vorwärts“ oder „Objekt bewegt sich in Richtung X“) und nicht mathematisch (x:= x + 5). Vor diesem Hintergrund wäre es also wünschenswert, wenn es in *Etoys* noch mehr natürliche Bewegungsoperationen gäbe. So wird ein Schüler durch die Anweisung „Zeichnung drehe Dich zu Punkt …“ bereits von komplexen geometrischen Berechnungen befreit, da der passende Drehwinkel automatisch ermittelt wird. Andere typische Bewegungsfunktionen und -bedingungen (z. B. das Verfolgen anderer Objekte, das Blockieren von Bereichen auf der Spielwiese usw.) müssen dagegen selbst implementiert werden.

Fazit

Der XO-Laptop unter SUGAR bringt von Haus aus gleich mehrere Entwicklungsumgebungen mit, die sich in Funktionalität, Mächtigkeit und Interaktivität deutlich voneinander unterscheiden. Für einen ersten Einstieg in die Programmierung von Grafiken ist das an der Schildkrötengrafik von LOGO orientierte Werkzeug *Turtle Art* durchaus geeignet. Mit SCRATCH kommen vor allem Multimedia-Elemente hinzu, die es erlauben, sehr motivierende Animationen und interaktive Ge-

schichten zu erstellen. Am mächtigsten ist sicherlich *Etoys*, da es letztlich das Programmieren in SQUEAK – und damit in SMALLTALK – erschließt. Hier gilt am ehesten die frühere LOGO-Maxime: „No treshold, no ceiling", also keine Einstiegshürde, aber auch keine Begrenzung nach oben. Da die Umgebungen nicht direkt aufeinander aufbauen, sollte man sich also gut überlegen, wie weit letztlich die Schülerinnen und Schüler geführt werden sollen. Ist das Ziel, am Ende informatische Grundkenntnisse erarbeitet zu haben, dann führt eigentlich kein Weg um *Etoys* herum. Bei *Turtle Art* und SCRATCH machen sich sonst zu schnell „Deckeneffekte" bemerkbar und die erworbenen Fertigkeiten sind nicht ohne Weiteres übertragbar.

Dr. Joachim Wedekind
Institut für Wissensmedien
Konrad-Adenauer-Straße 40
72072 Tübingen

E-Mail: j.wedekind@iwm-kmrc.de

M.Sc. Christian Kohls
Institut für Wissensmedien
Konrad-Adenauer-Straße 40
72072 Tübingen

E-Mail: c.kohls@iwm-kmrc.de

Literatur und Internetquellen

Allen-Conn, B.J.; Rose, K.: Fundamentale Ideen im Unterricht – Mit Squeak Mathematik und Naturwissenschaften verstehen (2008).
http://www.mttcs.org/Material/FundamentaleIdeen.pdf

Crickets:
http://llk.media.mit.edu/projects.php?id=1942

Freudenberg, R.; Hancl, M.; Mietzsch, E.: Es quiekt im Unterricht – Unterrichtstipps für den Einsatz von SQUEAK. In: LOG IN, 27. Jg. (2007), Heft 144, S. 30–38.

Freudenberg, R.: SUGAR – ein Betriebssystem zum Lernen. In: LOG IN, 29. Jg. (2009), Heft 156, S. 40–44 (*in diesem Heft)*.

Horn, M.S.; Jacob, R.J.K.: Tangible Programming in the Classroom with Tern. In: Begole, B. u.a. (Hrsg.): Conference on human factors in computing systems 2007 – CHI '07 Proceedings. New York: ACM Press, 2007, S. 1965–1970.
http://hci.cs.tufts.edu/tern/inter191-horn.pdf

Hyde, A. u.a.: Learning with Turtle Art. Floss Manual Turtle Art (2008).
http://en.flossmanuals.net/turtleart

iLearnIT.ch: Die Programmierumgebung Scratch (2009).
http://ilearnit.ch/de/scratch.html

Louden, K.C.: Programmiersprachen – Grundlagen, Konzepte, Entwurf. Bonn: Thomson, 1994.

Pane, J.F.; Myers, B.A.; Ratanamahatana, C.A.: Studying the language and structure in non-programmers' solutions to programming problems. In: Int. J. Hum.-Comput. Stud., 54. Jg. (2001), H. 2, S. 237–264.

Peppler, K.A.; Kafai, Y.B.: Creative Coding – Programming for personal expression (2005).
http://scratch.mit.edu/files/CreativeCoding.pdf

Romeike, R.: Animationen und Spiele gestalten – Ein kreativer Einstieg in die Programmierung. In: LOG IN, 27. Jg. (2007), H. 146/147, S. 36–44.

Rusk, N.; Resnick, M.; Berg, R.; Pezalla-Granlund, M.: New Pathways into Robotics – Strategies for Broadening Participation. In: Journal of Science Education and Technology, February 2008.
http://web.media.mit.edu/~mres/papers/NewPathwaysRoboticsLLK.pdf

Schiffer, S.: Visuelle Programmierung – Potential und Grenzen. In: Mayr, H. C. (Hrsg.): Beherrschung von Informationssystemen. Schriftenreihe der Österreichischen Computergesellschaft, Band 88. München; Wien: Oldenbourg, 1996, S. 267–286.
http://www.schiffer.at/publications/se-96-19/se-96-19.pdf

Schiffer, S.: Visuelle Programmierung – Grundlagen und Einsatzmöglichkeiten. Bonn u.a.: Addison-Wesley Longman, 1998.

SCRATCH:
http://scratch.mit.edu/

Tauber, M.J.: LOGO. In: LOG IN, 3. Jg. (1983), Heft 3, S. 37–42.

Tern – tangible programming (2008).
http://hci.cs.tufts.edu/tern/

Zuckerman, O.; Resnick, M.: A Physical Interface for System Dynamics Simulation. Cambridge (MA, USA): MIT, 2003.
http://web.media.mit.edu/~orenz/papers/system_blocks_chi03.pdf

Alle Internetquellen wurden zuletzt am 20. März 2009 geprüft.