ACTA DIDACTICA NAPOCENSIA

Volume 3, Number 2, 2010

THE POWER OF SCRIPTING: DGS MEETS PROGRAMMING

Jürgen Richter-Gebert, Ulrich Kortenkamp

Abstract: In this article we demonstrate how the combination of a system for dynamic geometry with a freely programmable scripting environment can be advantageously used in teaching and research. We explain the reasons behind various design decisions that were made by us when designing the language *CindyScript* and give examples that proof how they lead to easy and understandable code that can be used in education. We give several concrete application scenarios of the language that was developed by the authors and seamlessly interacts with the dynamic geometry system Cinderella.

Key words: Dynamic Geometry, Programming

1. Introduction

During the past three decades systems for doing dynamic geometry on a computer became an important tool for mathematical teaching, research and publication. The historical development can be traced back to Ivan Sutherlands Sketchpad [20] as early as 1963 and went through several characteristic phases – each of them motivated by certain didactic goals and formed by technical limitations and a constant input of user feedback. When in the 1980's computer systems became more and more available to schools and private households systems like the early Cabri 1.0 [1] and Geometers' Sketchpad [4] were published (see [19] for the detailed treatment of the evolution of these two products). In these early versions they were entirely focused on elementary geometry and intentionally only allowed for the use of numbers and geometric measurements in an extremely restricted way. When these systems gained popularity it became apparent that measurements and calculations are an indispensable part of geometry education in current curricula and, as a consequence, features to handle these were added to dynamic geometry systems (DGS). In the 1990's, numerous different systems for doing dynamic geometry were developed at several places mainly differing by the supplied geometric primitive operations and some differences in the user interface and software ergonomics. A new phase began at the end of the 1990's when various DGS groups started to regard dynamic geometry as a mathematical discipline and unveiled characteristic mathematical problems and phenomena that underlie the dynamic movement of geometric scenarios. Main problems involved the treatment of elements at infinity, undefined or degenerate intersections, the construction of loci, and the quite deep problem of continuous vs. conservative behaviour of a DGS [8][15][16]. By the end of the century a number of such *third generation DGS* had emerged based on more sophisticated mathematical concepts. With them, also advanced concepts for measurements, calculations and function plotting entered the scenario and became an important tool (not only) in high-school teaching and also in areas slightly apart from pure geometry.

For us as developers of one of these third generation systems called *Cinderella* [14], the early 2000's offered a fascinating scenario of user feedback. On the one hand users (in school, industry and university) became more and more aware of the power of the "drag-and-change" scenario in a dynamic geometry program to visualize interrelations within geometric sketches (see [5] for a good

---

This paper is an expanded version of the presentation given to the second Computer Algebra and Dynamic Geometry Systems in Mathematics Education (CADGME) conference at the University of Linz, Austria, in July 2009.

Published: 1 June 2010.

summary of the school perspective). On the other hand people started to recognize that behind a dynamic geometry system there are serious mathematical techniques involved (like, for instance, Projective Geometry and complex analysis in Cinderella [15]). This in turn invoked more and more requests for interactive treatment of various scenarios from other parts of mathematics that are related to geometry but cannot be covered by the usual *construction sequence paradigm* [8] of dynamic geometry software. Many of the users' requests, like the construction of fractal structures, using algorithms from discrete and computational geometry, or the visualization of arbitrary functional dependencies were extremely reasonable and it was easy to imagine how they could benefit from the interactive manipulation possible in a DGS. However, essential algorithmic features were missing to satisfy these valid requests. Various development groups faced this issue with various tactics. For instance, the *boolean point* feature of Cabri 2.0 [9] allows for creating conditional visibility of geometric elements. Others (e.g., GeoGebra [3]) decided to provide a collection of specialized high-level operations (e.g., a boxplot diagram function) that can be used to have more advanced interactions that satisfy specific pre-defined needs.

For us it became more and more apparent that a hybrid approach that still retains all advantages of a dynamic geometry system but allows for much more flexibility would be a promising venue. The goal was to add a *programming environment* that allowed for creating advanced mathematical scenarios in the same simple way as it is possible for geometric sketches. At the same time, such a programming environment should remain consistent with all mathematical requirements that are part of the DGS's underlying theory (in particular the continuous tracing of geometric objects in complex ambient spaces forces seriously such restrictions – we will however not deal with this problem or its solution in this article, but refer to [16] instead). This article describes (part of) the scripting language *CindyScript* and its design. The language is an integral part of Cinderella [17] starting with version 2.0. In the following sections we discuss several design decisions and how they influence the use of the DGS in teaching and learning.

2. Language Design

The development of the language *CindyScript* was lead by several design principles that we formulated based on our experience with mathematics teaching, both with and without computer support. The language should…

- *Interact seamlessly* with the DGS part of the program, without having to overcome technical or syntactical hurdles.
- Be easy to learn, because students and teachers should be able to concentrate on the *mathematical content, not the mathematical tool.*
- Admit access to mathematical functions on a high level, such that the users can benefit from the *abstraction* mathematics offers.
- Run in *real time*, such that the effects of both changes in the code and interactive manipulation of elements are not only explorable, but can be *experienced.*
- Be fun to use.[1]

These top-level requirements induce several concrete design decisions that mainly shaped the language. In what follows we want to explain a few of them and give concrete code fragments that explain the use and style of the language.

[1] It may seem strange that *fun* is a design principle for a language, but it is not. As another example we mention Ruby. As Yukihiro Matsumoto, the inventor of Ruby, writes in the foreword to [21]: "I believe that the purpose of life is, at least in part, to be happy. Based on this belief, Ruby is designed to make programming not only easy, but also fun."

2.1. Implicit typing

"Programming" is a very demanding and sometimes scary task for many teachers, as we have learned from lots of teacher training courses with DGS. One of the problems that prevent them from using a programming language is the syntactical overhead that comes with the language specification. Often it is necessary to specify lots of additional information that is not inherent to the actual task, just to program even the simplest functionality. Declaration of variables, function typing, import of libraries, class definitions etc. make it difficult to get used to a language. There are several approaches to this problem; a common one is to use an advanced IDE (integrated development environment) that offers scaffolding for code and other tools. There are also IDEs that are designed for education, for example BlueJ, a Java IDE that enables students to create working programs a lot easier than by using the command line tools only [6]. On the other hand, these environments and their "magic" can create a feeling of subjection. We decided on another approach where the language itself carries as little syntactical overhead as possible, and expressions should look like common mathematical formulae. Also, simple functionality should be doable by simple pieces of code. One of the fundamental design decisions in this direction was not to use explicit typing of variables. A variable that is used in a CindyScript program may have any type of value. Its semantic interpretation in a concrete context depends on this type of the value that is only distinguished internally. A value can be of the following types

- **`number`** (this may be an integer, a real or even a complex number),
- **`string`**,
- **`boolean`**,
- **`list`** (lists may be also used as vectors or, with lists of lists, as matrices),
- **`geometric element`**

An evaluation of an expression depends heavily on the concrete type of the arguments. If the arguments and the operators are incompatible the evaluation may result in an undefined expression. Without additional syntactical overhead the code fragment

```
a=2; b=3; println(a+3*b);
a=[3,2]; b=[4,5]; println(a+3*b);
a="hello"; b=" world"; println(a+b);
```

results in the output

```
11
[15,17]
hello world
```

In the first line the two variables are added as a pair of numbers, in the second line they are added as a pair of lists (interpreted as vectors) and in the third line they are concatenated as a pair of strings.

2.2. Interaction with geometry

The interaction with the geometric part of the program should be as seamless as possible. For this reason variables whose names equal the name of a geometric element in a construction are predefined to be the (pointer to) the geometric element itself. At any time they may be overwritten with other values. So if a geometric construction contains four free points A, B, C, D the single line of code

```
D.xy=(A.xy+B.xy+C.xy)/3;
```

moves the point D to the centre of gravity of the other three points. The **`.xy`** operator accesses the Euclidean coordinates of the geometric elements. There is one language convention that makes it even possible to simplify this line. If geometric elements are used in arithmetic expressions they are replaced by their corresponding coordinates automatically. With this, the above line becomes

```
D.xy=(A+B+C)/3;
```

It is also possible to access other characteristic properties of geometric elements. So for instance the piece of code

```
D.color=if(D.x>0,red(1),blue(1));
```

conditionally sets the colour of point D depending on its x-coordinate.

2.3. Realtime requirements and Events

It is one of the fundamental features of dynamic geometry systems to allow for direct interaction with a mathematical object (usually a geometric sketch). Every mouse action should be translated directly into a response of the mathematical object. The same realtime-interaction paradigm should also hold in the presence of scripts. This requires that the language interpreter is fast enough to evaluate the script during every single screen refresh (or even more often). It furthermore requires a detailed control on the moment of execution of a script, since different scripts may be associated to different semantic occasions (initialisation, screen refresh, mouse action, etc.). For this reason we introduced a simple event model that triggers the execution of the scripts. Every script is associated to a particular occasion. Putting the above script for instance in the "Draw" event (compare Figure 1) results in the behaviour that the script is executed always immediately before a screen refresh. The position of point D is updated dynamically when A, B or C are dragged, and D's colour is changed dynamically.

Using the semantics of various events it is possible to vary the behaviour of a dynamic sketch in a very granular way. In particular, using scripts that are sensible to various specific mouse actions are very powerful for enhancing a sketch by providing additional feedback for the user.

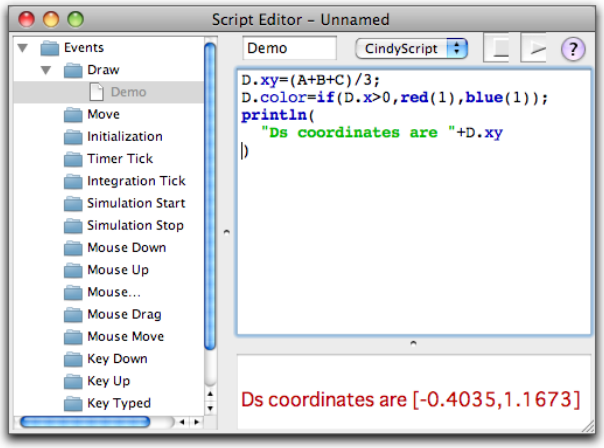


Figure 1. The script editor in Cinderella

2.4. List operations

Lists constitute an extremely powerful data structure in CindyScript that can be used in various contexts. A list is a sequential collection of elements of arbitrary type. The members of a list may be, for instance, numbers, strings, geometric elements, or again lists. The intended semantics of a list in a concrete programming context may be simply a list of objects, a vector, a set or other similar collections. Lists of numbers may be treated as vectors. Two such vectors may be added if their length is compatible. A vector may also be multiplied by a scalar. Lists of vectors of the same length are treated as matrices. Arithmetic expressions concerning matrices and vectors are interpreted in the obvious way. So, for instance the code fragment

```
a=[[1,2],[3,2]]*[3,2];
```

calculates the product of a matrix and a vector and stores the result in the variable a. By this it is quite easy to perform geometric transformations by matrix multiplications.

The built-in functions **`allpoints()`** and **`alllines()`** return lists of all geometric points or lines in a drawing. Similar operations are available for other types of objects.

Besides arithmetic operations, lists may be processed in various ways. Operations for iterating through lists **`forall(…)`**, applying functions to them **`apply(…)`**, conditionally selecting elements **`select(…)`**, sorting lists **`sort(…)`**, etc. are available. As an example consider the following code fragment. It traverses all points of a geometric construction and selectively sets the colour depending on the x-coordinate.

```
pts=allpoints();
forall(pts,p,
  p.color=if(p.x>0,red(1),blue(1));
);
```

In the next section we will see more advanced usages of lists. Before this we have to consider a few very fundamental language paradigms.

2.5. Functional programming

There are various principal ways how a programming language can be designed. Roughly speaking a language could be sequential (like *Basic* or *Pascal*), object oriented (like *C++*, *Smalltalk*, *Ruby*, or *Java*), functional (like *LISP* or *Mathematica*), logic based (like *Prolog*), or a blend of these (see the standard book [18] for a much more detailed description). While in the first two paradigms the emphasis is on advanced control and data structures, the paradigms of the last two are influenced by mathematical structures. Usually sequential and object oriented languages need a lot of syntactical overhead. Functional programming is by far closer to evaluation of mathematical computations than logic based programming. The programming paradigm of CindyScript is mainly driven by the functional philosophy. This allows for programming structures that are comparatively close to mathematics. Still, the more traditional procedural programming style is also available.

As a rule of thumb one could say that in CindyScript every operator is a function. Thus it takes arguments as input, evaluates them according to certain rules (perhaps with some side effects like setting the position of a point) and finally returns a value. Even statements that may look procedural at first sight are implemented in a functional fashion. We will explain this by the example of the **`if`**-operator. This operator takes three arguments: the first must be a boolean value, the other two are programs (in fact the programs are themselves treated as functions and hence will return a value). The semantics of the **`if`**-operator is as follows: If the first operator evaluates to **`true`** then the program given as the second operator is evaluated and its value is returned, otherwise the program given as the third operator is evaluated and its value is returned. Thus the conditional assignment of a colour to a point as we did in Section 2.2 could be either coded as.

```
if(D.x>0,
  D.color=red(1),
  D.color=blue(1)
);
```

(this is in a sense the procedural programming style), or as we did earlier in a more functional style:

```
D.color=if(D.x>0,red(1),blue(1));
```

Combining functional programming with list operations has an incredible expressive power. We demonstrate this in a little piece of code that calculates the first n prime numbers

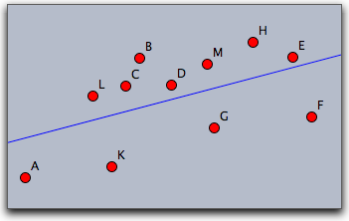
```
divs(t):=select(2..t/2,i,mod(t,i)==0);
primes(n):=select(1..n,i,divs(i)==[]);
print(primes(100));
```

The first line defines a function **`divs(t)`** that returns a list of all divisors of **`t`** except for **`1`** and **`t`** itself. It does so by traversing all numbers in the list **`2..t/2`** and selecting those that divide **`t`** without remainder. The second line defines a function **`primes(n)`** that selects all prime numbers of size at most **`n`**. It does so by traversing the list **`1..n`** and selecting all those **`i`** for which **`divs(i)`** returns the empty list, i.e. **`i`** has no non-trivial divisors.

2.6. High level mathematics

Many high-level mathematical operations are directly accessible via CindyScript. Vector and matrix operations, typical linear algebra primitives, finding roots of polynomials are only a few of them. It would be too tedious to list all of them here.[2] Instead of that we will give a simple usage scenario that uses some high-level math functions to calculate the regression line of all points in a geometric drawing (see picture alongside the code).

```
pts=allpoints();
A=apply(pts,p,(1,p.x));
b=apply(pts,p,(p.y));
m=transpose(A)*A;
v=transpose(A)*b;
erg=inverse(m)*v;
plot(erg*(1,x));
```

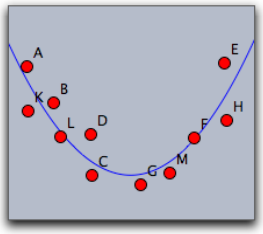


The code is a direct translation of the usual mathematical procedure to calculate a regression line.[3] The first line creates a list of all n points (x_i,y_i) in the drawing. The second line uses the **`apply`**-operator to create a matrix A with n rows $(1,x_i)$. The third line calculates the vector b of all y-coordinates. With this preparation the two parameters r, s of the regression line $y = r + sx$ can be calculated as

$$(r,s) = (A^TA)^{-1}(A^Tb).$$

This is exactly what is done in the next three lines of code. The last line plots the regression line using the fact that **`erg*(1,x)`** calculates the scalar product of two vectors. With only marginal change the same code can also be used to do for instance quadratic regression:

```
pts=allpoints();
A=apply(pts,p,(1,p.x,p.x^2));
b=apply(pts,p,(p.y));
m=transpose(A)*A;
v=transpose(A)*b;
erg=inverse(m)*v;
plot(erg*(1,x,x^2));
```



2.7. Fast prototyping and direct feedback

A language like CindyScript does not make programming of complicated algorithms an all easy business, but it enables users to realise simple things in a really simple manner. Often already a few lines of code may add fundamentally new functionality to a geometric scenario. An important design decision in the development of CindyScript was to have the language as an integral component of the DGS. Every small bit of code should be able to create an immediate visual and behavioural feedback in an already existing geometric drawing. By this it is possible to write programs in a "fast prototyping" style where one has a constant feedback loop of the code and its effect. This is particularly important for teaching purposes since students can learn mathematics by programming in a quite intuitive way. The spirit here is similar to quite classical approaches to teaching programming like the *LOGO* language of Seymour Papert [12] where direct visual feedback was achieved by the use of a "Turtle Graphics" drawing device. Programming tasks similar to LOGO exercises can also be given in CindyScript, however, the microworld goes far beyond the turtle graphics approach of the 80's.

[2] The complete documentation of CindyScript is available at http://doc.cinderella.de

[3] Incidently, the mathematical description of this can be found in Appendix F of [20]

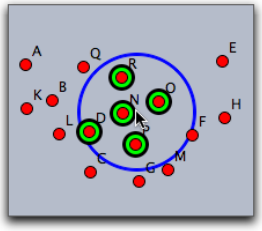
3. Usage scenario

In this section we are going to study several scenarios in which the combination of CindyScript and the DGS Cinderella is appropriate. We will not explain every single line of code but rather concentrate on the overall programming philosophy and program development strategy.

3.1. Enhancing geometric sketches

A typical situation arises when some small piece of functionality has to be added to a geometric scenario. Highlighting a geometric element under certain conditions, adding a slider, creating some mouse-over functionality are typical applications. The following piece of code will (when put in the *mouse move*-event) highlight all points that are in a certain radius of the mouse pointer. It will also draw a circle with this radius around the mouse.

```
clrscr();
pts=allpoints();
m=mouse().xy;
rad=2;
drawcircle(m,rad);
sel=select(pts,p,|p,m|<rad);
forall(sel,p,draw(p,size->13));
repaint();
```

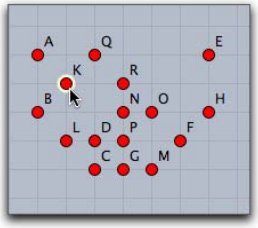


The first and the last line are required to refresh the drawing, since the piece of code is not put in the default drawing event slot, where this is done automatically.

3.2. Changing the behaviour of the DGS

The default behaviour of a dynamic geometry program is not always suitable for a certain interaction scenario. For instance, it might be required that if a point gets close to a circle it snaps to it or that all points automatically snap to a nearest grid point while dragging them. The following extremely simple program realizes this snap functionality:

```
pts=allpoints();
forall(pts,p,
  p.xy=round(p);
);
```

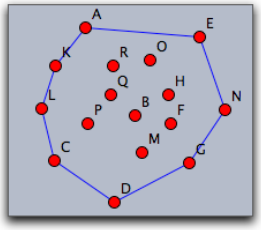


In the third line the *xy*-coordinate of each point is replaced by its rounded version. When put in the *draw*-event this script immediately influences the point positions while dragging.

3.3. Algorithmic Geometry

Much more demanding programming tasks arise in the context of computational geometry. In contrast to usual applications of dynamic geometry systems combinatorial and algorithmic tasks dominate in this field. Still, it is highly desirable to visualize the concepts by interactive drawings. With the combination of a DGS and scripting it is relatively straight-forward to create such demonstrations. We will exemplify this by a slightly more advanced task: finding the convex hull of a cloud of points. A segment (*A*,*B*) in the convex hull is characterized by the following criterion: All other points of the point cloud lie on the same side of the segment (*A*,*B*). Using list operations we can directly encode this criterion. The following piece of code does the job.

```
pts=allpoints();
left(a,b):=select(pts,p,area(a,b,p)~<0);
right(a,b):=select(pts,p,area(a,b,p)~>0);
segs=pairs(pts);
hull=select(segs,s,
  or(left(s_1,s_2)==[],right(s_1,s_2)==[]);
);
drawall(hull);
```

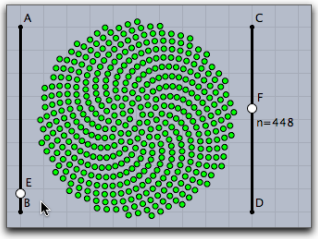


This code needs a little explanation. The third and the second line define functions that return lists of points that are either entirely left or entirely right of the segment given by the points **a** and **b**. The sidedness decision is made by the function **area(a,b,p)** that returns the oriented area of a triangle. The comparison **~<** tests for "being surely smaller" and is equivalent to **<-eps** with a small positive number **eps**. The line **segs=pairs(pts)** creates a list of all pairs of points. From this list all elements of the convex hull are selected and drawn (see picture alongside the code).

3.4. Programming user interfaces

Another interesting application of scripting arises when a geometric sketch (or some other visualized scenario) should be accompanied by a user interface to control several options of the visualization. The possible applications of scripted user interfaces are only restricted by imagination. We here want to give one extremely simple and two high-end examples. An extremely simple example arises when one needs a slider in a geometric drawing. Sliders are often accompanied with very problem specific coordinate transformations, rounding, snapping to values, etc. The following code is the basis of a small visualization that visualizes the location of seeds in a sunflower.

```
d=.1°*(|B,E|/|B,A|-0.5)+1;
w=137.508°*d;
n=round(|D,F|/|D,C|*800);
repeat(n,i,
  p=0.2*sqrt(i)*(sin(i*w),cos(i*w));
  draw(p);
);
drawtext(F+(0.2,-.8),"n="+n);
```



The lines of code marked in red, create internal parameters from a small geometric construction that mimics two sliders. For instance **n=round(|D,F|/|D,C|*800);** calculates the numbers of seeds that have two be drawn from the position of the three points *D*, *F*, and *C*.

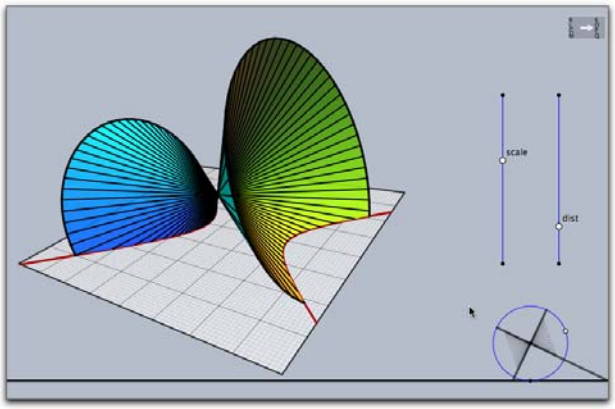


Figure 2. Creating conic sections from a 3D cone

A more advanced scenario is shown in Figure 2. This is, a demonstration applet that explains how ellipses, hyperbolas and parabolas are generated as sections of a double-cone. Scripting is used to create a 3D position control of the image, a circular slider for the position of the double cone with respect to the cutting plane, and two plain sliders.

Figure 3 below shows a drawing of Pappos's Theorem that is combined with a collection of scripted buttons that demonstrate the combinatorial symmetries of the configuration. Pressing a button triggers a continuous transition that permutes the elements indicated by the button. This and the above example are part of a huge collection of interactive sketches accompanying the book *Geometriekalküle*. The

sketches can be found at http://www.geometriekalkuele.de which is part of the even larger collection Mathe-Vital at http://www.mathe-vital.de. Almost all content available there uses scripting.

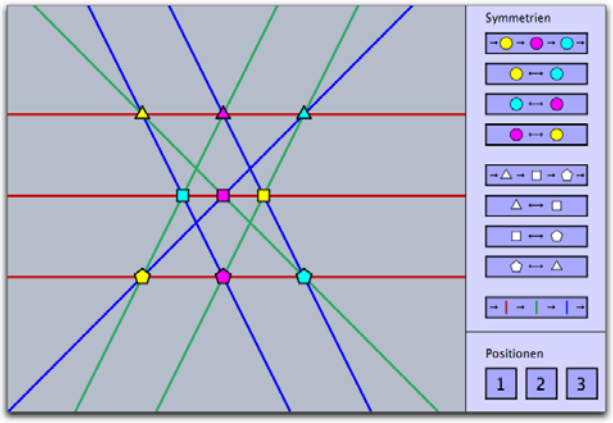


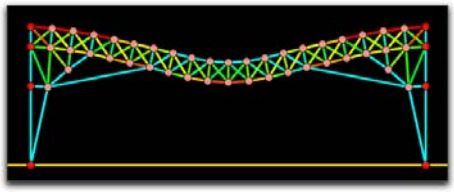
Figure 3. Advanced customization of the user interface

3.5. Interfacing to physics simulations

The current release of Cinderella also possesses the ability to do physics simulations essentially by simply "drawing an experiment." Since the physical parameters of the objects (masses, forces, velocities, charges, energies, etc.) are fully accessible by the scripting engine this opens plenty of interesting and often highly instructive usage scenarios for scripting.

We exemplify this again with a small example. Consider a physics simulation of the statics inside a bridge. A bridge may be considered as a network of very stiff springs. Each such spring is either compressed or stretched by the forces in the bridge. The following script associates to each spring a colour that indicates the amount of compression or expansions (i.e. the inner force of the corresponding bar).

```
springs=allsprings();
f(x):=hue(max((min((0.5,x+0.25)),0.0)));
forall(springs,s,s.color=f(s.ldiff));
```



The second line specifies a colour function that associates to the length differences a rainbow colour change between *red* (full extension) and *cyan* (full compression). The third line loops through all springs in the drawing. Again the changes are even visualized dynamically if the bridge is shaken by a mouse action. Amazingly, the comparably short and simple script enhances the interactive demonstration a lot.

4. Teaching Programming with CindyScript

So far we only mentioned scripting in the context of enhancing mathematical sketches by adding functionality. But scripting (i.e. programming) itself is a highly interesting intellectual activity worth to be taught. There are still no standard methods for teaching computer programming, and it is beyond the scope of this article to give a full introduction to the different research perspectives.

The reason why one would teach/learn programming is twofold. One reason is to know a specific programming language to be used for a specific applied task. The other reason is in our opinion by far more important and fundamental. Programming is an intellectual activity that forces to structure ones thoughts. It needs and enhances the ability for *formal reasoning* – the foundation of all scientific and technological thinking [13].

For the novice, learning to program is often accompanied with specific difficulties. In particular, in most languages a lot of syntactic overhead has to be overcome until the first piece of working code is written. The situation gets even worse if the beginner wants to create a program with graphical output and mouse interaction as input. This is one of the reasons why languages and environments like Squeak [2] or Logo (for recent developments we want to mention Netlogo) have been developed. While these environments are perfect for early (and also not-so-early) education in computer programming, they are not perceived as solutions for real problems the students are facing.

A DGS with scripting facilities is another solution to this problem, and it comes with the additional advantage to be useful for the solution of mathematical problems that the students may encounter, e.g. in modelling activities. Advanced graphical output is inherent to every DGS and mouse interaction directly conforms to the drag-and-change metaphor of dynamic geometry. By this, such a hybrid setup (like the Cinderella/CindyScript-tandem) offers the possibility to learn programming in a very clear and at the same time highly educational and highly motivating environment.

Our experiences with high school and university students are that in this environment even absolute beginners can create interesting projects within a very short time. The examples we gave throughout this article where just a few lines of code were sufficient to produce highly interesting and non-trivial effects demonstrate this. We also refer to [9].

We want to conclude this section with a short report about a course for beginners in programming that demonstrates how far novices can get within only two days. In a guided exercise the students were first introduced into simple drawing techniques that were used to draw a “smiley”. In a second stage the features of the smiley (hair, smile, eyes, etc.) were made controllable by sliders. A simple data structure was then used to introduce a kind of “genome” for these features and to implement parent/child-inheritance functionality. On the second day the students were introduced to physics simulations and learned to program a simple swarm-simulation (about ten lines of code). Then the two projects were combined to get a swarm of smileys that mated and got little smiley-children. So within two days the students were able to program a complete and extendable evolutionary population dynamics.

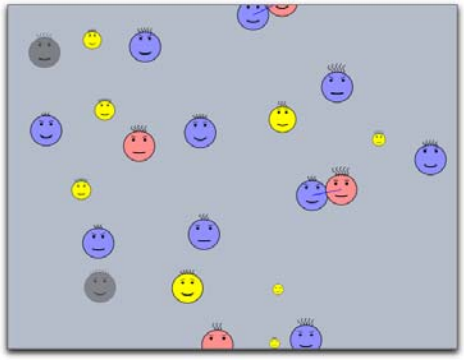


Figure 4. Snapshot of a smiley population

5. Conclusion

The scripting language presented in this article has been designed as an easy-to-learn yet powerful tool to enhance the functionality and the user interface of Dynamic Geometry software. In order to enable students, teachers and researchers to express their mathematical thoughts in a straightforward way the syntax of the CindyScript language is close to regular mathematical notation. The interaction with all parts of the software, including simulation features, is possible. Therefore, we hope that this tool can be used at all levels of mathematics education, starting even in K-12 levels.

Literature

[1] Baulac, Y., Bellemain, F. & Laborde, J.M. (1988): Cabri-Géomètre, un logiciel d'aide à l'apprentissage de la géomètrie. Logiciel et manuel d'utilisation, Cedic-Nathan, Paris. Software. See http://www.cabri.com.

[2] Cardelli, L. & Pike, R. (1985): Squeak: A language for communicating with mice. In: Proceedings of the 12th annual conference on Computer graphics and interactive techniques. ACM SIGGRAPH Computer Graphics, vol. 19, issue 3, p. 199-204.

[3] Hohenwarter, M. (2002): GeoGebra. Software. See http://www.geogebra.org

[4] Jackiw, N. (1991): The Geometer's Sketchpad. Berkeley, Calif.: Key Curriculum Press. Software. See http://www.keypress.com

[5] King, J., & Schattschneider, D., eds. (1997): Geometry turned on: Dynamic software in learning, teaching and research. Washington, DC: Mathematical Association of America.

[6] Kölling, M., Quig, B., Patterson, A. and Rosenberg, J. (2003): The BlueJ system and its pedagogy, Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology, Vol. 13, No. 4, Dec 2003.

[7] Kortenkamp, U. & Richter-Gebert, J. (1998): Geometry and Education in the Internet Age. In Ottmann, T. & Tomek, I. (eds.): Proceedings of ED-MEDIA 98, Freiburg: AACE.

[8] Kortenkamp, U. (1999): Foundations of Dynamic Geometry, Ph.D. thesis, ETH Zürich.

[9] Kortenkamp, U. & Fest, A. (2009): From CAS/DGS integration to algorithms in educational math software. The Electronic Journal of Mathematics and Technology. Vol. 3, No. 3. See https://php.radford.edu/~ejmt/ContentIndex.php.

[10] Laborde, J.-M. & Bellemain, F (1993): Cabri-Geometry II. Software. Texas Instruments. See http://www.cabri.com

[11] Mayer, R.E., ed. (1988): Teaching and Learning Computer Programming: Multiple Research Perspectives. Hillsdale: Lawrence Erlbaum Associates.

[12] Papert, S. (1980): Mindstorms: Children, Computers, and Powerful Ideas. New York: Basic Books.

[13] Pea, R. & Kurland, M. (1984): On the cognitive effects of learning computer programming. New Ideas in Psychology 2:22, 137-168, Elsevier.

[14] Richter-Gebert, J. & Kortenkamp, U. (1999): User Manual for The Interactive Geometry Software Cinderella, Springer-Verlag, Heidelberg.

[15] Richter-Gebert, J. & Kortenkamp, U. (2001): Grundlagen Dynamischer Geometrie. In: Zeichnung – Figur – Zugfigur (German). Henn, H.W., Elschenbroich, H.J. & Gawlick, Th. (eds.). Hildesheim, Berlin: Franzbecker. Available online at http://kortenkamps.net/papers/2001/DGOW1.pdf.

[16] Richter-Gebert, J. & Kortenkamp, U. (2002): Complexity issues in Dynamic Geometry. In: Foundations of Computational Mathematics (Proceedings of the Smale Fest 2000). Cucker, F. & Rojas, J.M. (eds.). World Scientific.

[17] Richter-Gebert, J. & Kortenkamp, U. (2006): The Interactive Geometry Software Cinderella, Version 2.0. Software. See http://cinderella.de

[18] Roy, P.v. & Haridi, S. (2004): Concepts, Techniques, and Models of Computer Programming. MIT Press.

[19] Scher, D. (2000): Lifting the Curtain: The Evolution of The Geometer's Sketchpad. The Mathematics Educator, Vol. 10, No. 1. Available online at http://math.coe.uga.edu/TME/v10n2/4scher.pdf

[20] Sutherland, I.E. (1963): Sketchpad: A Man-Machine Graphical Communication System, Technical Report No. 296, Lincoln Laboratory, MIT, Boston, MA. Available online at http://www.dtic.mil/cgi-bin/GetTRDoc?AD=AD404549&Location=U2&doc=GetTRDoc.pdf

[21] Thomas, D. & Hunt, A. (2001): Programming Ruby. Addison Wesley Longman. Available online at http://www.ruby-doc.org/docs/ProgrammingRuby/html/index.html.

Authors

Jürgen Richter-Gebert, TU Munich, Zentrum Mathematik, Germany, e-mail: richter@ma.tum.de

Ulrich Kortenkamp, Centre for Educational Research in Mathematics and Technology (CERMAT), University of Education Karlsruhe, Germany, e-mail: kortenkamp@cermat.org

Biographical Notes

Ulrich Kortenkamp is full professor for Mathematics and Education at the University of Education Karlsruhe, Germany. He is working at the interface of mathematics, computer science, and education. His primary research interests are in Interactive Geometry and its applications to teaching and learning.

Jürgen Richter-Gebert is full professor for “Geometry and visualization” at the Technical University Munich, Germany. His special interests include automatic theorem proving, dynamic geometry, geometric programming, user interface design and many more.

Both authors developed the geometry software Cinderella, whose first version was available in 1999. The software received many awards, among them the European Academic Software Award and the Deutsche Bildungssoftwarepreis. The collection *Mathe Vital* which was created mainly with Cinderella under heavy use of the scripting facilities was awarded the MedidaPrix 2008.