38. From simple turtles to interactive objects for visual mathematics

Ivan Kalas

Department of Informatics Education, Comenius University
842 48 Bratislava, Slovakia

kalas@fmph.uniba.sk

Abstract

In our short paper we explore and further elaborate the possibility of Imagine to specify the shape of a turtle as a list of simple Logo instructions. This option offers surprisingly broad palette of interesting and challenging applications where you can create miscellaneous shapes which Imagine rotates automatically to express the turtles' actual headings.

In our Minerva CoLabs project we go even further: Due to the mechanism of partial evaluation of the drawing lists we create dynamic shapes the look of which depend on several external values and conditions. One of the goals of the project is to specify and create rich set of highly interactive Imagine objects to represent fractions (and later probably also other concepts and relations in elementary mathematics).

In our paper we examine how such visualizations of mathematics objects should be created so that afterwards they can be used by the educators to build stimulating environments for learning fractions by exploring and discovering.

Keywords

Programmable shapes of turtles, drawing lists, Imagine, new cultures of learning

1. Introduction

Shapes of Logo turtles developed dramatically during last two decades. Initially, the shape of a turtle was a simple outline of a small turtle or a triangle, which had several rotations, built in the core of Logo engine to display the actual heading of the turtle. Today, turtles are often used as either stationary or complex animated characters of interactive compositions created by children. Our two versions of Logo, SuperLogo in 90s and Imagine since 2001, have considerably contributed to that process.

In SuperLogo we have introduced a new data object called image, which can be used (beside other options) as a shape of a turtle. The decision to work with images in exactly the same way as with other data objects (i.e. with words and lists) has proved to be very creative and powerful – in SuperLogo images became regular first order data. An image is a sequence of frames, which can be interpreted by the turtle in two different ways:

- For traditional turtle frames of its shape serve to display its actual heading. There is an automatic mechanism in SuperLogo to choose and display the corresponding frame for the actual heading.

- If the user declares a turtle to be *animation one*, the connection between *frames* and *actual heading* is broken and it is up to the user to decide, when the turtle will display which frame. In this way the tool gives him/her the possibility to create the illusion of a movement, as used in the animation films, see Figure 1. Simple commands of the language allow the user to give a command with the meaning *display the following frame*.

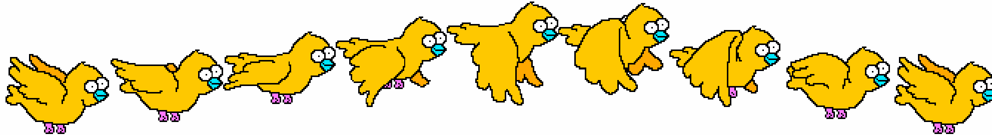

Figure 1. Image is a sequence of frames, which can create the illusion of a movement

2. Imagine and Shapes ff Turtles

Imagine has made another step in this development, namely in two different directions:

- We decided to remove the distinction between traditional and animation turtles. Therefore we generalized the concept of image in the following way: Each image is a sequence of frames, which are always interpreted as different pictures for displaying different headings of turtles. Each frame, however, may consist either of one or of a sequence of frame items, which perform like an animated GIF: if the frame consists of only one frame item, the shape is static. If it consists of several items, Imagine will automatically alternate them (using included time delays). Thus the engine automatically creates and controls the illusion of animation.
- The second direction is also quite innovative. For defining the shape of a turtle, you can use Logo language itself. All programmable shapes, i.e. shapes specified in this way stick to the relative frame and Imagine rotates them according to actual headings of turtles.

Introducing animation turtles in SuperLogo has obviously increased its power because it made it possible and easy to create animated compositions and stories, where the goal is the process itself, not its final result (as usual in traditional activities). In spite of that we have always felt two disadvantages in connection with images and animations. The first of them has disappeared by moving from SuperLogo to Imagine, the second one hasn't:

- The user – when creating such compositions – had to define his/her own main animation control loop. This isn't needed in Imagine any more because its own independent process can control each object. The generalized structure of images with frames and frame items shifted the task of controlling animations to the engine. For example, to create a turtle, set to it the shape of a flying bird and let it fly from left to right forever is now trivial.
- To place the animated turtles into your composition requires that you either create corresponding images or get them from somewhere. Both ways are difficult and too much demanding. Users are once again forced to rely on libraries of professional images.

We have tried to reduce the disadvantage (2) in Imagine by introducing rather interesting option: When Logo beginners learn how to type in a sequence of instructions or define simple drawing procedure (like house, tree, boy, girl, star…), let us show them a way how to create new shapes for turtles using exactly the same palette of tools, that is, merely by typing in instructions and defining simple procedures for drawing. At the moment when they are able to draw a square (e.g. in a thick red pen) by saying:

```
? setPC "red setPW 8
```

```
? repeat 4 [fd 60 rt 90]
```

they can use the same piece of Logo program to specify a square shape for the turtle. They simply enclose their piece of program into square brackets – thus creating their first *drawing list* – and use it as the input to *setShape* command:

```
? setShape
    [setPC "red setPW 8
       repeat 4 [fd 60 rt 90]]
```

What will happen? The red thick square will become the turtle's new shape. It wont be drawn in the background picture, it is a turtle that lives above the background. Notice that there is a fundamental difference between these two squares:

1. The second square is a turtle; that is, it is ready *to be moved* by simple turtle commands like forward and back. It lives above the background; the square shape hasn't become a part of the background picture. It can as well be moved by the direct manipulation tools.
2. If the turtle's shape is specified by a *drawing list*, it can *automatically* be rotated. If you say rt 10 it will react quite intuitively – the shape will be redrawn in new actual heading. If, for example, it is a tree, you can easily make it bend from one side to another by left and right commands, see Figure 2.

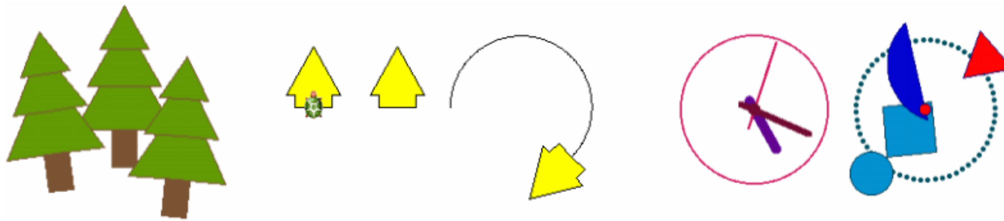

Figure 2. Shapes of turtles can be programmed in Logo using drawing lists. In this way you can create miscellaneous attractive shapes, which are automatically rotated by Imagine itself

3. Drawing Lists and Programmable Shapes

Shapes of turtles can be programmed using drawing lists, which can be used as inputs to the setShape command. In (Kalas, Blaho 2000) we specified what drawing lists were. We also presented a broad palette of possibilities offered by this mechanism, together with a sequence of topics, which teachers and their students may explore to learn how to utilize this feature. Note however that the language that can be used in drawing lists is restricted to the following Logo commands:

- pen and fill settings by setPenColour, setPenWidth, setFillColour etc.,
- pen modes by pu and pd,
- movements by forward and back,
- turnings by right and left,
- control structure repeat,
- points, dots, circles, filled circles, ellipses, filled ellipses by corresponding Imagine primitive commands,

- setting position in Cartesian coordinates by setPos, where actual position of the turtle is temporarily considered to be the origin [0 0],
- setting heading by setHeading, where the starting heading of the turtle (heading at the time when the drawing list starts to be interpreted) is temporarily considered to be heading 0,
- specifying font by setFont and printing a text by label,
- drawing a filled polygon by primitive command polygon drawing-list as a part of the bigger drawing list,
- specifying a spline curve by drawSpline drawing-list as a part of the bigger drawing list,
- including a sequence of points by outline list-of-points as a part of the bigger drawing list.

A drawing list can be assigned to a turtle as its shape using the setShape command or can be used by the drawList, polygon, drawSpline, putPicture, setPicture for buttons, glide or stampPicture commands, or can serve as an input to the operations toPoints and toVectors.

Drawing lists prove to be a strong tool in the hands of a more advanced developer as well, because they allow him/her to quickly and easily create very complex shapes, polygons or splines without any considerable demands of a big memory space.

4. Programmable Shapes in Visual Mathematics

All shapes, which we explored so far, are useful and inspiring alternatives to more traditional turtles' shapes. However, it is also helpful to concentrate on certain well-defined topic and explore in depth how programmable shapes can contribute to its microworlds, that is, creative computer laboratories in which children can explore and construct their understanding.

In (Kalas, Blaho 2003) we concentrated our attention to mathematics education and we examined how Imagine and its new enhancements support understanding in modern exploratory mathematics, which increasingly employs creative computer environments. We examined whether Imagine can stimulate the emergence of new cultures for constructing, exploring and understanding. We claim that Imagine helps to make mathematical objects and relations between them tangible. We base this conviction on five key properties of Imagine, namely:

- programmable pictures,
- direct manipulation tools,
- events,
- object oriented structure and
- parallel independent processes.

We also presented a selection of well-known visible mathematics activities and matched them to properties and techniques of Imagine, which makes it possible to implement them. In the following chapters we will further examine the topic of visual mathematics, however we will in detail elaborate the possibility of programmable shapes to dynamically react on changing external values and conditions.

5. Dynamic Shapes

When you define a new instance or a new class in Imagine, you have to specify the *parent object* and a *list of settings* of the form [setting value setting value …]. This list has two special yet very important features. If a value in this list is a word prefixed by a colon : or if it is enclosed in brackets (…), it will be evaluated by Imagine – as a *variable* in the first case, *or a piece of computation* in the second case, and only then used as a value for its setting. Similar mechanism of *partial evaluation* is also possible in a drawing list if you use it as an input to setShape command. However, in such case you have to initiate this mechanism by putting ! in front of the list. As an example we can set the shape of a turtle to be an orange square with the size expressed by a variable a:

```
? setShape ![setFC "orange polygon [repeat 4 [fd :a rt 90]]]
```

5.1. Dynamic shape expressing actual value or values

The possibility of partial evaluation within the programmable shape of a turtle offers surprising potential. We can for example create a turtle with the shape which changes in accordance with certain dynamically evolving quantity or quantities. As an example, such turtle may have the shape of a small rectangular card with a text, which shows to the actual distance of that turtle from the origin. Let the turtle's autoDrag setting be true and its pen penUp. If we specify its onDrag event as myShape [0 0] and we define myShape command as in Figure 3, then our turtle-card can be dragged within the page and will always show its actual distance from the origin. Note that to find the distance between the turtle's actual position and a point P we take advantage of (a) the possibility to interpret points as vectors, and (b) process them as first order data, that is, apply subtraction to get the vector from P to pos and apply abs to get the magnitude of that vector.

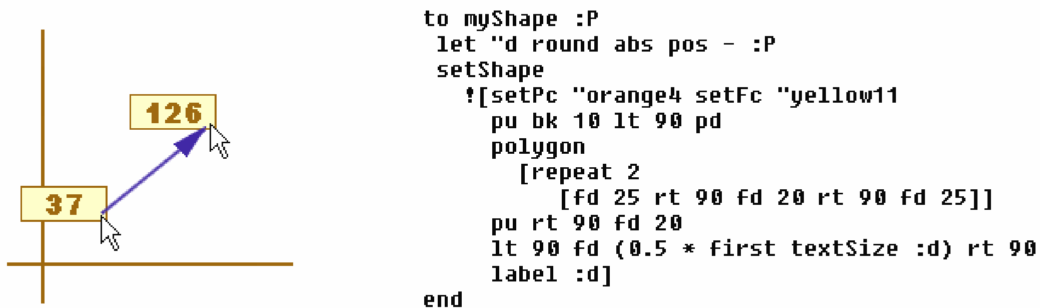

Figure 3. myShape command first computes the distance d, then specifies the shape inside the setShape command as if it was drawn by a turtle. The pen colour and fill colour are set, then the pen backs from its initial position by pu bk 10 lt 90 pd and draws a card - filled polygon sized 50 times 20. Note that the position (hot spot) of this shape will be the initial point of the pen, that is, in the centre of the card. Finally, the actual distance d is added atop the card by the label command

5.2. Dynamic shape being indirectly modified by other control gadgets

Turtle-card of the previous example keeps its size constant, only its label (i.e. actual information about the distance) is being updated repeatedly while we drag the card. In other situations it may be useful to modify the programmable shape's size as well. We will illustrate such situation by turtle t2 and its diameter setting with an initial value of 50. To create such user-defined setting, we either open the last but one tab of its ChangeMe dialogue and press the Add button there, or in the command line we type the instruction t2'haveOwn "diameter

50. Note that henceforward turtle t2 will recognize two new procedures: the diameter operation to get its value and the setDiameter command to modify it. Now, let us define myShape command for t2, see Figure 4. It sets the shape of t2 to be a filled circle with the diameter specified by the actual value of the diameter setting. Let us add a vertical slider next to t2. Its role will be to set the actual value of t2's diameter (setting and shape as well). To do so, right click the slider and open its ChangeMe dialogue. There we specify its onChange event as:

```
t2'setDiameter value t2'myShape
```

where value refers to actual value of the slider.

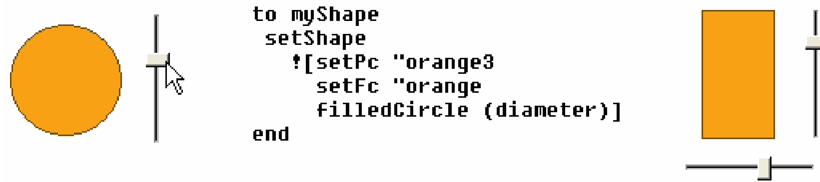

Figure 4. This example illustrates how we can modify the shape of the turtle "from outside". When we reset the slider, the actual value of t2's diameter is reset accordingly and t2 is asked to update its shape

Figure 4 illustrates also another example when one turtle has several gadgets to modify its programmable shape: vertical slider specifies the height of that turtle' shape while horizontal slider specifies its width.

5.3. Dynamic shape being directly modified by the mouse

The approach which we have just adopted has one important disadvantage in many situations. Namely, each dynamically changing object needs its own control gadget or even several gadgets. Such indirect control is inconvenient if we have many objects in parallel, if we want to clone them easily etc. To avoid this and increase the direct interactivity of our circular object, we would prefer to:

a. drag it by the mouse – as a whole – along the page, if we click its inside,

b. make it smaller or bigger, if we click it near the border and drag.

To implement such complex dragging, the simple and standard autoDrag mechanism won't be sufficient any more. We need to create our own version of dragging based on the onDrag event. Let us set the onDrag event of our circular object to be user-defined command myDrag:

```
to myDrag
  setPos mousePos-clickedPos
end
```

where clickedPos is a standard Logo operation which outputs the vector whose initial point is the current position of the turtle and whose terminal point is the position of the most recent mouse click within the turtle's shape. Its function in the myDrag command is to compensate for the displacement of our initial click within the turtle's shape, see left part of Figure 5.

Next step is to define the onLeftDown event for our circular object. As its reaction, our own decide command will be run once. It will analyze the initial position of the mouse cursor and decide whether it belongs to the inside of the circle or to its border. If the first option is true, decide will set the onDrag reaction to be myDrag, otherwise it will set mySize to be the

reaction. To set the reaction, decide makes use of the standard setEvent command. To distinguish the inside from the border, the following condition is used:

```
diameter / 2 - abs clickedPos > 4
```

If the difference between radius (i.e. diameter / 2) and the magnitude of clickedPos (i.e. the distance of the mouse cursor from the centre) is bigger than let's say 4, decide classifies the click to take place inside the circle. Number 4 specifies how wide is the zone which decide considers to belong to the border.

```
to decide
 ifElse diameter / 2 - abs clickedPos > 4
   [setEvent "onDrag [myDrag]]
   [setEvent "onDrag [mySize]]
end
to mySize
 setDiameter 2 * abs mousePos - pos
 myShape
end
```

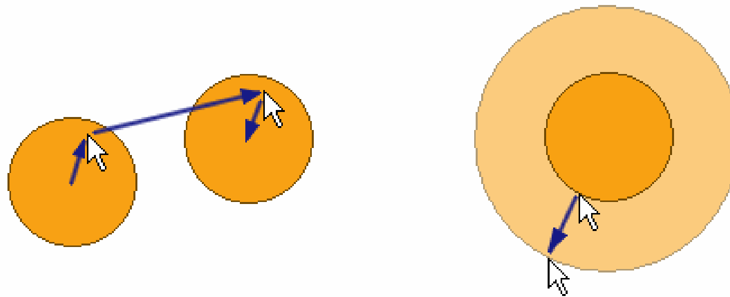

Figure 5. Drawing lists with partial evaluation can be used to implement complex onDrag behaviour. If you click the circular object by its inside and drag, you will drag the whole object. If you click it close to its border and drag, you will make the circle smaller or bigger

6. Dynamic Shapes for Exploring Mathematics

At present we are exploring the potential of dynamic shapes and trying to take advantage of them also within the Minerva CoLabs project (see the project's web page at http://matchsz.inf.elte.hu/colabs/). One of our goals there is to identify a rich set of visual representations of fractions within mathematics curriculum of children aged 8 to 12 and implement them as highly interactive objects in Imagine. In this way we want to develop a building set for mathematics educators to help them create modern, attractive and efficient teaching/learning materials for children. We believe that within such visual interactive microworlds children will explore concepts and relations between fractions in an intuitive and tangible style. We also believe that our building set of fraction objects will serve as an authoring tool for educators with modest or no knowledge of programming.

Obviously, different representations of the same fractions will be used in parallel to support the learning process. Among them we will definitely need a "traditional" textual visualization with nominator and denominator, see Figure 6. Several of these objects may be used within one screen, illustrating various values and relations or presenting problems to be solved,

observations to be made etc. Let us explore how such object can be implemented in Imagine making use of programmable pictures and other properties.

Each "textual" fraction object we want to build will be represented by a turtle with a programmable shape. Each of these objects will be in one of two possible modes: in a user mode (see the leftmost part of Figure 6) or in an author mode (see all other parts of Figure 6). When in the author mode, the author or developer of the educational material can:

- set its size (i.e. visual size of the shape, not the values of nominator and denominator) by dragging the control element 1 upwards or downwards,
- set whether the object will visualize corresponding fraction in reduced form or not by checking or clearing small control box 2,
- specify which fraction object within the screen will be visualized by this textual representation. To specify this, the author clicks the small box 3 and drags it (see element 4) as an anchor along the screen and releases it atop another fraction object, see rightmost part of Figure 6 and also left part of Figure 7,
- drag the whole object by its inside along the screen.

To toggle between the user mode and author mode, one has to click the object by the mouse with Shift + Ctrl keys pressed at the same time.

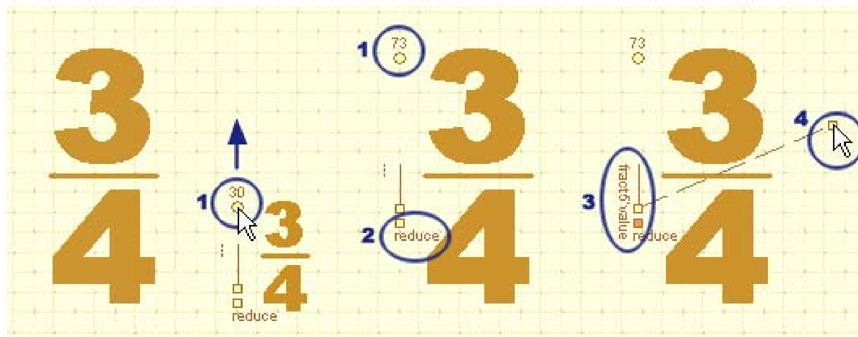

Figure 6. We see four objects of the class FractionA. They are turtles with several specific settings, programmable shapes and high level of interactivity

Although it is more difficult to create such complex objects than previous dynamic objects we have seen in chapter 5, it is feasible due to programmable shapes, events and parallel processes in Imagine. Objects in Figure 6 are instances of the class FractionA, which in turn is a subclass of the class Turtle. Beside all standard settings, these objects have several specific ones like:

- the font to write (within the shape) the word reduce or the name of the fraction being visualised by this object (see small vertical label number 3 in Figure 6),
- the font to write the nominator and denominator of the object, that is, what will the size and look of the object be,
- actual values of the nominator and denominator,
- is the object locked, that is, is it in the user mode or in the author mode,
- is the object reduced, that is, are the actual values shown in reduced form or not.

The key command – and the most complex – is the myShape command of the FractionA class. Using the setShape […] structure it builds all parts of the shape according to all actual values of all specific settings. When the object is currently locked, its control elements (see letters 1, 2 , 3 and 4 in Figure 6) are not created within the shape at all. If the object's anchor is at present being dragged to another object, the myShape command will have to create the temporary flexible line between the object and its anchor.

When in the "quiet" mode (that is, when not being dragged, resized, anchored etc.) the object has only two events attached to itself. The onLeftDown event runs our own decide clickedPos command (similar to one which we built in chapter 5 although more complex). Its role will be to decide – according to actual clickedPos – what will happen when we start dragging the object by the mouse. The onLeftUp event will run our own myLeftUp command, which will correctly finish the process of pulling the anchor, see below.

The decide command analyzes the clickedPos vector, that is, the relative displacement of the initial mouse cursor position within the shape itself. Thus it distinguishes, which control element of the shape has been clicked (if any at all).

```
to decide :disp
 toFront
 ifElse <to lock / unclock>
   [setLocked? not locked?
    myShape handle
    setEvent "onDrag []]
   [ifElse <to reduce / unReduce>
      [setReduce? not reduce?
       myShape handle
       setEvent "onDrag []]
      [ifElse <to pull the anchor>
         [setEvent "onDrag [myDefine]]
         [ifElse <to make it smaller / bigger>
            [setEvent "onDrag [mySize]]
            [setEvent "onDrag [myDrag]]]]]
end
```

Note again that the clickedPos (the displacement) is analyzed only because the decide command needs to know what reaction to the following onDrag events should be specified by the setEvent command. Namely:

- if we click the object (together with Shift + Ctrl keys pressed) to toggle between the user mode and the author mode, decide will reset the mode, update the object's shape and engage nothing (i.e. empty list) as the reaction to onDrag,
- if we click the "check square" <to reduce / unReduce>, decide will accept this change, update the object's shape and engage nothing (i.e. empty list) as the reaction to onDrag,
- if we click small square <to pull the anchor>, decide will engage our own myDefine command as the reaction to onDrag. myDefine will control the process of pulling the anchor. It will also update the shape itself until the anchor is released,

- if we click the element 1 <to make it smaller / bigger>, decide will engage our own mySize command as the reaction to following onDrag events,
- otherwise, decide will engage myDrag command to run dragging the whole object as it is along the screen.

The reaction to onLeftUp event is our own myLeftUp command. It processes the situation when we have been pulling the anchor and now we release the left mouse button. (In all other situations myLeftUp does nothing.) If there is no fraction object (of any available representation) bellow the anchor when released, no dependency (link) is established. If we released the anchor atop a fraction (see the leftmost part in Figure 7), our connectMeTo that_object command is run, which starts a new monitoring process to establish one way dependency between our fraction and that_object. This process will continuously observe the other object (its nominator and denominator) and will update the shape of our fraction whenever any change happens.

7. Final Considerations

In our CoLabs project we are building different representations of fractions making use of similar techniques and mechanisms, see Figure 7. Obviously, there are many important problems to be examined yet – both on the technical and educational levels. One of such technical problems to tackle is how to manage and efficiently organize numerous dependencies within one running application, see (Tomcsanyi 2003).

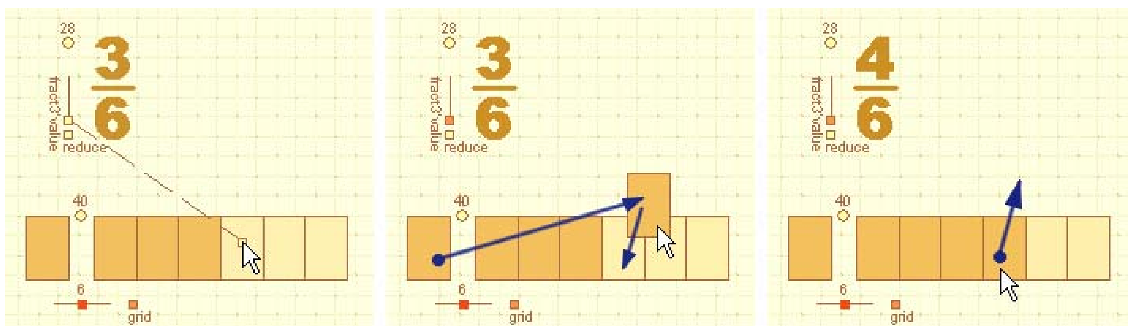

Figure 7. Alternative representation of fractions through a fraction box. textual fraction objects can be connected to such box. User may add parts into the box or remove them from the box and observe also the corresponding (dependant) textual representation – either reduced or not.

All the objects we have mentioned so far somehow represent certain values. However, we believe that our approach can be generalized to create other objects to visualize:

- *dependencies*, which may for example express relations like: Turtle t2's position is always the same as t1's position but displaced by the vector [100 0], or Turtle t2's position is always in the middle between actual positions of turtles t1 and t3,
- *generators*, which may – in specified frequency – transmit certain values (information) to specified object or objects. Such generator could for example every second set turtle t6's pen width to be the next item within the cycle of possible values 5, 6, 7, 8, 9, 10, 5, 6...
- *processes*, which may – in specified frequency or under specified conditions – run certain block of instructions for an object or objects.

If we manage to create such rich and complete sets of visual mathematics blocks (objects) with high level of interactivity, we may help to provide children with open laboratories to explore and discover new concepts and relations between them, we may help to create new cultures of learning – more intuitive, attractive and tangible.

8. References

Kalas I and Blaho A (2000), *Imagine... a new generation of Logo: Programmable Pictures*, Proc. of Conference on Educational Uses of Information and Communication Technologies, IFIP 16th World Computer Congress 2000, pp. 427 - 430, ISBN 3-901882-07-3, China

Kalas I and Blaho A (2003), *Exploring visible mathematics with Imagine: Building new mathematical cultures with a powerful computational system*, Marshall G and Katz Y (eds), Learning In School, Home And Community, Kluwer Academic Publishers, Boston 2003, pp. 53 - 64

Tomcsanyi P (2003), *Implementing object dependencies in Imagine*, short paper submitted to EuroLogo 2003, Porto