39. Implementing object dependencies in Imagine Logo

Peter Tomcsanyi

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

tomcsany@internet.sk

Abstract

In our short paper we try to examine programming tasks where relations exists among objects. It means that an object's attributes depend on some attributes of other object(s). We make some observations and based on them we develop several general classes, which can be useful when programming these kinds of tasks. We also use these classes to implement some non-trivial examples including a small but useful subset for dynamic geometry.

Keywords

Imagine Logo, Object-oriented programming.

1. Introduction

Logo can be used to program a variety of tasks. In this paper we would like to focus on tasks where the state of one object depends on the state of other object(s).

A simple example of such a task is to create a program containing two turtles where one of them can be moved programmatically or dragged freely by the mouse and the other one should follow the same movements like in a mirror. It means that the painting made by the second turtle will be a mirror image of the painting made by the first turtle.

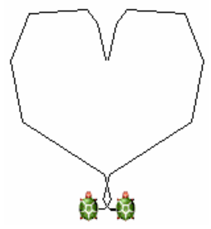
2. Implementing dependencies by running additional processes

If we chose the y-axis to be the mirror, the name of the first turtle is t1 and the name of the second turtle is t2, then the relation between t1 and t2 could be expressed as:

```
t2'pos = list -t1'xcor t1'ycor
```

As in Imagine there can run several parallel processes, the straightforward idea of implementing such a relation is converting it to a process, which runs forever:

```
forever [t2'setpos list -t1'xcor t1'ycor]
```



A bit bumpy picture as a result of using `forever`

Now let's allow t1 to be dragged by the mouse:

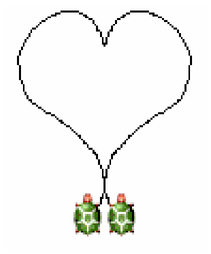
```
t1'setAutodrag "true
```

And we can try to create a symmetrical drawing. You can see that the program in principle does what we wanted. But, depending on the speed of your computer and the version of Windows, you may experience that the movement of both turtles is not smooth and the drawings are bumpy as you can see it on our picture.

This happens because the process launched by the `forever` command consumes too much time to re-position t2 each time even if t1 has not moved at all. Also other programs running in the same time may suffer from the fact that Imagine consumes too much time for actually doing nothing but re-position t2 to the same place as it was before.

Let's stop our process and start another one:

```
every 10 [t2'setpos list -t1'xcor t1'ycor]
```

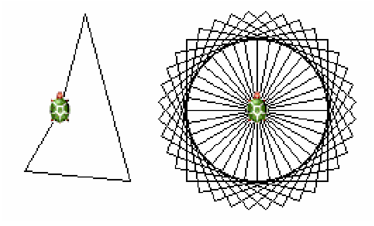


A smoother picture when using `every`

This process does the same as the first one, but it does not try to do it all the time. It just re-positions t2 every 10ms. Now t1 can be dragged much more smoothly even on slow computers and t2 is still able to follow it with enough precision.

Now let's try to paint something by Logo commands:

```
repeat 36 [repeat 4 [fd 50 rt 90] rt 10]
```



The dependent turtle cannot follow each movement of t1

You can see that t2 is not able to follow quickly enough t1. And this will not improve even if we switch back to the previous code and use `forever` instead of `every`.

We can conclude: It is easy to rewrite a relation into a program using one process per each relation to be kept. But this way of implementing relations is limited to situations when the state of the followed object does not change too quickly or when it does not matter that the dependent object will not react on each of the change of the other object. A good example of such a situation is when the mouse drags one object and other objects should follow it in a certain way.

3. Intercepting position changes for turtles - the posTurtle class

In the previous section we have seen that if t2 tries to follow t1 "blindly" without any synchronisation then it may miss some positions of t1 if it moves quickly. Therefore we need another way to implement dependency so that each relation is enforced only at the time when the values used in that relation change.

In our simple example it means that t2 should be repositioned only when the position of t1 changes. We would need to have a kind of onPosChanged event, which would be triggered in t1 each time when its position changes.

Imagine Logo has no such built-in event. So let's implement a new class of turtles, which trigger an onPosChanged event whenever the position changes.

At first let's think when a turtle changes its position:

after executing the commands forward, back, setpos, setxy, setxcor, setycor, home...

when it is dragged by mouse (with autodrag setting on),

after it is moved using the right-click menu's Move function, when its page or pane is cleaned by the cs command, when its page or pane is resized,...

It is quite easy to intercept the position changes listed under points 1. and 2. Changes listed under the point 3. are not so easy to intercept because these are executed internally by Imagine Logo interpreter and no turtle command is actually executed. For our examples and many other uses these commands need not to be intercepted, though.

Let's create the new class names posTurtle and implement its changed setpos procedure:

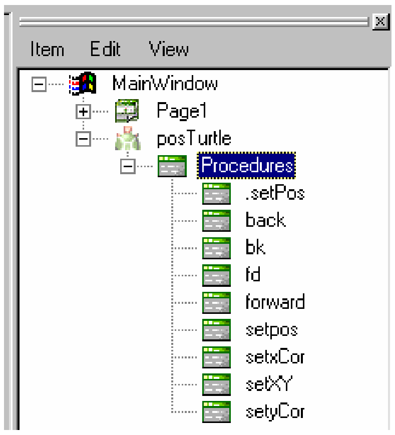
```
newclass "turtle "posTurtle []

posTurtle'to setpos :xy

 let "oldpos pos

 usual'setpos :xy

 runevent "onPosChanged

end
```

You can see that the setpos procedure was changed in such a way that it stores the current pos to a local variable :oldpos, then it calls the usual setpos and then an event named onPosChanged is triggered. Thanks to dynamic scope of Logo the body of the event can use the :oldpos variable if it needs to know the previous position.

In the class posTurtle we can re-program also the commands like forward, fd, back, bk exactly in the same way as we did it for setpos. This way we can solve each of the commands listed in point 1.

Now let's try to intercept position changes, which result from dragging the turtle by mouse. At first glance it seems to be complicated because no setpos is called. But there is an undocumented command .setpos. It is called each time the turtle is re-positioned due to dragging by mouse. Knowing this, it is easy to intercept .setpos calls in the same way as we did for setpos:

```
posTurtle'to .setpos :xy
 let "oldpos pos
 usual'.setpos :xy
 runevent "onPosChanged
end
```



The posTurtle class with all its procedures defined

This way we can finish all the procedures of the posTurtle class as you can see in figure.

Turtles of this class will have a new event onPosChanged and each such a turtle can react on it adding its own code while all the complexities of how this event is implemented stay hidden from the programmer.

Let's re-implement our symmetry example using posTurtle. Erase all the turtles, then create one posTurtle (it will get the name t1) and one usual turtle (t2):

```
eraseobject all
new "posTurtle [autodrag true pos [20 20]]
new "turtle []
```

Now we need to define the onPosChanged event of t1:

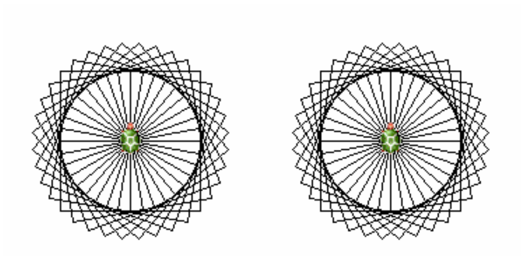
```
t1'setevent "onPosChanged [t2'setpos list -t1'xcor t1'ycor]
```

To enable users defining the onPosChanged event not only programmatically but also using the user interface, we need to add its name to the eventNames list. For turtles this list is defined in the eventNames common variable of the Turtle class. We can copy the content of the list to class posTurtle and add the new event's name on its beginning:

```
posTurtle'havecommon "eventNames fput "onPosChanged turtle'eventNames
```

You can see the difference that t2 can follow also programmed movements of t1:

```
repeat 36 [repeat 4 [fd 50 rt 90] rt 10]
```



A posTurtle can follow the movements exactly

If we wanted to intercept other changes of the turtle (like its heading or any other state variable), then we could define a new event and then a new set of procedures intercepting some calls, which change the desired state variable and then invoke the newly defined event.

4. Programming relations from the correct direction – the depObject class

When we defined the relation, then we said that t2 should mirror the movements of t1. It means that we have defined the relation from the viewpoint of t2. But in the previous section we have programmed it from the opposite direction. We have added code to object t1 to update t2 each time when t1's position changes.

In very simple projects it is not a problem. But for more complex projects it would be better to be able to program the relation from the same direction as it was naturally defined. It means that if we say that object X depends on objects Y and Z then the code implementing the dependence should be inside X, not inside Y and Z.

We could achieve this goal by implementing another class of turtles, which will be based on posTurtle and which will implement the relations from their technical direction. But not only turtles will may need to use the mechanism of evaluating dependency. And even not all turtles, which would need that mechanism, need to be posTurtles. Therefore we will develop an abstract class having all necessary code and later we will attach this object to any other class or object as its additional behaviour using the mechanism of behaviours, which is one of remarkable features in Imagine.

```
newclass "Main "depObject [dependList []]

depObject'to dependsOn :li

 ; defining turtles which this one depends on

 haveown "deflist []

 let "na myname

 foreach "ob (se :li)

   [if and word? :ob not number? :ob

     [ask :ob [setdependlist fput :na dependlist]

      setdefList lput :ob defList

     ]
```

```
      ]
    adjustDep
  end
```

```
depObject'to adjustDep
  ; this procedure is called whenever the object needs
  ; adjusting because other objects, which it depends on,
  ; were changed
  ; it's body is empty, descendants need to define it
end
depObject'to objChanged
 ; this object has changed therefore adjust all dependent objects
 ; descendants of this class must call objChanged when any of the
 ; settings, which we are interested in, change
 askeach dependlist
   [adjustDep]
end
```

The class depObject holds names of related objects in dependList and whenever the object is changed it asks for updating all dependent objects by calling the procedure adjustDep of each such object. So each object or class must define its own adjustDep, which defines its dependency on other objects. And this is what we wanted to achieve - the code, which defines the dependency, is located in the dependent object.

Now we can create a class named posDepTurtle based on posTurtle having the behaviour depObject attached and calling its procedure objChanged on each change of its position:

```
newclass "posTurtle "posDepTurtle []
posDepTurtle'addBehaviour "MainWindow'depObject
posDepTurtle'setEvent "onPosChanged [objChanged]
```

Each turtle, which has some dependencies defined has another list named defList, which holds the list of object names which it depends on.

As for now we only deal with objects, which can be dependent only on position of another objects therefore we implemented the onPosChanged event as a call to objChanged procedure. If we wanted to add other kinds of dependency then we would need to call objChanged each time when we know that a particular setting has changed.

Now we can try to re-implement our simple example: at first we delete all turtles and create t1 as usual, just now it will not be a posTurlte, but a posDepTurtle instead.

```
eraseobject all
new "posDepTurtle [autodrag true pos [20 20]]
Now we create t2 - it is also a posDepTurtle:
```

```
new "posDepTurtle []
```

And we define the dependency by defining the adjustDep procedure of t2. To achieve a general solution, its body does not refer to t1, but instead it refers to "the first object I am linked to" expressed in Logo it is `first defList`:

```
t2'to adjustDep

 setpos list -ask first deflist [xcor] ask first deflist [ycor]

end
```

And finally we define the dependency. T2 should depend on t1:

```
t2'dependsOn "t1
```

If you look at the code above you can see how easily we can use posDepTurtle instances. The "free" ones just need to be an instance of posDepTurtle and the dependent ones must define the kind of dependency in their adjustDep procedure and also which other turtles they depend on by calling dependsOn.

5. An example: Chained turtles

We want to create a chain of turtles. The first one (the "head") can be dragged by mouse and all other turtles in the chain will follow it keeping a predefined distance from each other. We also want the head to turn into the direction where we drag it.

The chain will consist of turtles having the same behaviour. Therefore we at first define a class named followTurtle. It is based on posDepTurtle because it must react on pos changes and it also needs to have the behaviour depObject attached.

`followTurtle` has a new state variable called dist. This variable holds the distance, which this turtle should keep from its predecessor in the chain. The default value is set to 50, but each instance can have its own value.

```
newclass "posDepTurtle "followTurtle [rangestyle window dist 50]

followTurtle'addBehaviour "MainWindow'depObject
```

Now we define the dependence in adjustDep procedure. The turtle should always turn towards its predecessor in the chain and should go such a distance that it will end in the distance of dist from that turtle:

```
followTurtle'to adjustDep

  seth towards first defList

  fd (abs (ask first defList [pos]) - pos) – dist

end
```

Now we can create the head turtle. We redefine its onPosChanged to turn its heading to the correct direction. We must not forget to execute the inherited onPosChanged event:

```
new "followTurtle [pen pu autodrag true rangestyle window name head]
```

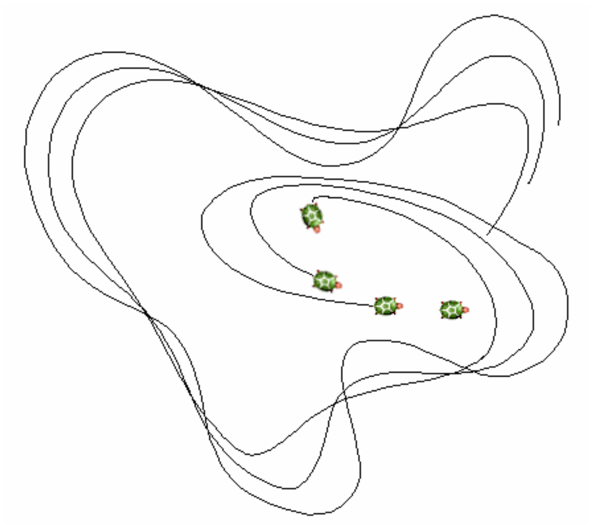
```
head'setEvent    "onPosChanged   [setheading   180+towards   :oldpos   run
myParent'event "onPosChanged]
```

Now we can create the first turtle after the head and two other turtles in the chain:

```
new "followTurtle []

t1'dependsOn "head

repeat 2 [new "followTurtle [] lastobject'dependsOn word "t repc]
```

We can drag the head turtle and all the whole chain will move and draw. We can also achieve different distances between turtles in the chain by changing the value of dist variable in each turtle:

```
askEach butmember "head all [setDist 20+10*bf myName]

head'objchanged
```



A drawing created by the chain of four followTurtles

The chain of turtles now behaves like connected with rods of fixed size. We can try to create another kind of turtles, which will be chained loosely - like with a rubber band. To achieve this we define another class of turtles where adjustDep procedure will not reposition the turtle immediately to its final position, but starts a process, which will move the turtle from the current position to the new one with decreasing speed:

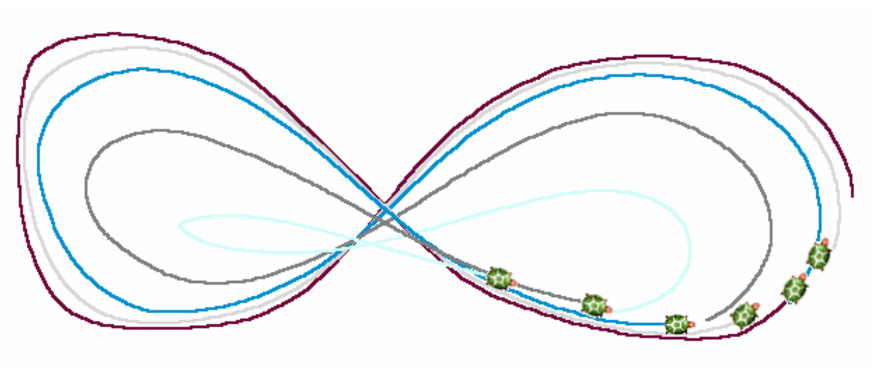
```
newclass "followTurtle "looseFollowTurtle [k 0.1]

looseFollowTurtle'to adjustDep

 if done? word "p myname

   [(every 10 [step] word "p myname)]

end

looseFollowTurtle'to step

 seth towards first defList
```

```
    let "di (abs (ask first defList [pos]) - pos) - dist
    if :di < 0.5 [stopme]
    fd :di*k
  end
```

You can see that the step procedure moves closer the turtle towards its predecessor in the chain by going k times the distance between the turtle and its predecessor where k must be a variable of the turtle having a value between 0 and 1. If the distance is too small then the process of approaching the predecessor stops. The adjustDep procedure asks if there is a process running for this turtle and if no then it launches one.

Now we can add two looseFollowTurtle instances to our chain. The first one will keep the distance of 60 pixels having k on the default value of 0.1 and the other one will keep a distance of 70 pixels having k=0.05.

```
new "loosefollowturtle [dist 60] lastobject'dependsOn "t3
new "loosefollowturtle [dist 70 k 0.05] lastobject'dependsOn "t4
askEach all [setpc any setpw 2]
```



A drawing created by the chain of four followTurtles and two looseFollowTurtles

6. An advanced example: dynamic geometry

Unlike GeomLand or Elica, Imagine has no built-in features, which would especially support creation of geometric objects along with their dependencies. In this section we would like to show that the basic mechanism of dependent objects developed in previous sections could enable us to implement a basic set of dynamic geometry objects. And we will also improve our basic depObject class to deal much more effectively with situations where one turtle depend on several other turtles.

The basic object for all geometric shapes will be geomShape. It is a descendant of posTurtle because some of its descendants may need the onPosChanged event introduced in posTurtle. geomShape also needs the behaviour depObject.

```
newclass "posTurtle "geomShape [pen pu rangeStyle window shapeTemplate
[]]
geomShape'addbehaviour "MainWindow'depObject
```

Procedure updateShape updates the current shape based on the value of the variable shapeTemplate. The shape of each geometric object will be defined by a drawing list containing Logo commands. updateShape also adds the object's name into its shape at the position defined by the function namepos. The shape must be updated for the first time when a new object will be created therefore we define an onCreate event.

```
geomShape'to updateshape
 setshape ! se shapetemplate [pu setpos (namepos) label (myname)]
end
geomShape'setevent "onCreate [updateShape]
```

The main mechanism of dependency in geometric shapes will not be based on position changes, but on shape changes. Therefore adjustDep for each object must update its shape and call objChanged to propagate the changes along links.

```
geomShape'to adjustDep
 updateshape
 objChanged
end
```

Point is the simplest geometric shape. Its shapeTemplate defines a small circle having black border and the inner colour is its pen colour.

```
newclass "geomShape "point [common'shapeTemplate [setpc black setfc ( pc
) filledcircle 6] common'nameprefix p stamping false]
```

Points can be of two kinds. The first kind is a free point, which can be moved. To create it just call `new "point [...]` with nothing special inside the brackets. Other points will be defined as intersections of two objects and must be created by calling `new "point [intersect [ob1 ob2] ...]` where ob1 and ob2 are names of objects. The special word `intersect` will not define a new variable of that name as other words on the initialisation list do. We test and delete it from the list it inside the Create procedure of the point class using the getparam primitive. After actually creating the new object we can use the information gathered from the list to define the dependencies of the new object and the variable deftype, which will hold the type of the point.

```
point'to create
 let "int getparam "intersect []
 usual'create
 ifelse not empty? :int
   [haveown "deftype "int pu dependson :int
    make "initList se :initList [pc grey]
   ]
   [haveown "deftype "free
```

```
     make "initList se :initList [pc red autodrag true]

    ]

  end
```

The adjustDep procedure adjusts the position of the dependent point and, if its stamping variable is set to true, stamps its shape on each such change. By calling usual'adjustDep we achieve that for intersection points also all dependent objects will be updated (because geomShape'adjustDep contains a call to objChanged). Free points need to update their dependent objects on each position change therefore we must call objChanged from their onPosChanged event:

```
  point'to adjustDep

   case deftype

     [int [setpos computeIntersection first defList last defList]]

   if stamping [stamppicture ! shapetemplate]

   usual'adjustDep

  end

  point'setevent "onPosChanged [if deftype = "free [objChanged]]
```

The namePos function defines the position where to write the object name:

```
  point'to namepos

    op list 7 (item 2 textsize myname)/2

  end
```

The linear object is the base for segments and straight lines. It defines two new variables: pt1 and pt2. They will hold two points defining the linear object.

```
  newclass "geomShape "linear [pt1 [0 0] pt2 [0 0]]
```

Linear objects can be defined in two ways. By giving two points (`new "segment [points [p1 p2]]`) or by giving one point and one other linear object and the new objects should be created going trough the given point and being perpendicular to the given linear object (`new "line [perp [p1 s3]]`). Both these cases are tested in the Create procedure. The adjustDep procedure adjusts the linear object according to its type and the namePos procedure defines the posision where to show the name of the object (in the middle between pt1 and pt2). The function perpendicular gets one point and computes another point on a line, which is perpendicular to the object and contains the given point.

```
  linear'to create

   let "pts getparam "points []

   let "perp getparam "perp []

   usual'create

   ifelse not empty? :pts
```

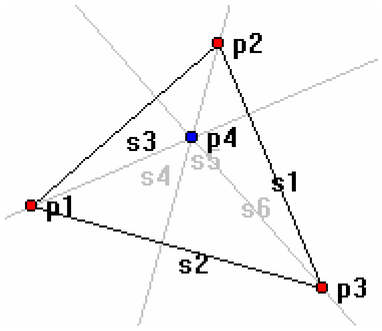
```
    [haveown "deftype "points
     dependson :pts
    ]
    [if not empty? :perp
      [haveown "deftype "perp
       dependson :perp
      ]
    ]
  toback
 end
 linear'to adjustDep
  case deftype
    [points [setpt1 ask first defList [pos]
             setpt2 ask last defList [pos]]
     perp   [setpt1 ask first defList [pos]
             let "p pt1 setpt2 ask last defList [perpendicular :p]]
    ]
  usual'adjustDep
 end
 linear'to namepos
  op pt1+(pt2 - pt1)/2
 end
 linear'to perpendicular :pt
   let "a (last pt2) - last pt1
   let "b (first pt1) - first pt2
   let "c (last pt1)*((first pt2)-first pt1)-
          (first pt1)*((last pt2)-last pt1)
   let "t (-((:b*last :pt) + (:a*first :pt) + :c)/(:a*:a+:b*:b))
   let "res :pt + list :a*:t :b*:t
   if abs :res - :pt < 1e-10
     [make "t :t+0.001 op :pt + list :a*:t :b*:t]
   op :res
 end
```

Now we can define the two linear objects: segment and line:

```
  newclass "linear "segment [common'namePrefix s common'shapeTemplate [pu
  setpos (pt1) pd setpos (pt2)]]

  newclass "linear "line [common'nameprefix s common'shapeTemplate [pu
  setpos (pt1-(pt2-pt1)/(abs pt2-pt1)*1000) pd setpos (pt1+(pt2-pt1)/(abs
  pt2-pt1)*1000)]]
```

Now we can try to construct a kind of canonical construction of dynamic geometry: a triangle with three of his heights and the point where they intersect:

```
  new "point [pos [-214 -105]]

  new "point [pos [-49 115]]

  new "point [pos [123 -61]]

  new "segment [points [p2 p3]]

  new "segment [points [p1 p3]]

  new "segment [points [p1 p2]]

  new "line [perp [p1 s1] pc LightGrey]

  new "line [perp [p2 s2] pc LightGrey]

  new "line [perp [p3 s3] pc LightGrey]

  new "point [intersect [s4 s5] pc blue]
```



The resulting construction

After creating this construction we can play a bit with it by dragging the red points. You can see that the whole construction keeps together thanks to the defined relations between objects. This is how dynamic geometry systems should work.

By examining the construction a bit more carefully, you can see that the blue point p4 moves in a bit bumpy way. You can visualise this if you write:

```
  p4'setStamping "true
```

You will see that the path of p4 is a zigzag line instead of a smooth curve. Also other elements of the construction may refresh quite slowly and bumpily on a slower computer. The first idea may be that Imagine is too slow for such a task. But examining a bit further we can understand that our program updates p4 several times per each position change of any of the red points. If p1's position changes then it updates s2, s3 and s4. The update of s4 updates p4 for the first time. The change of s2 changes s5 and it updates p4 for the second time.

And finally change of s3 changes s5 and it updates p4 for the third time. So for one movement of p1 there are three movements of p4 and only the final one puts it to a correct place from geometrical point of view.

The big number of updates could slow down larger constructions to an unusable level even on quick computers and it also causes problems by moving some elements in a scattered way so that the Stamping feature of points cannot be used to visualise the correct shape of its path.

To get rid of both problems we must redesign partly the class depObject, namely its objChanged procedure. It should not just blindly call adjustDep of all objects on dependList rely on the fact that updating those objects will cause updates of other dependent object in a recursive way. Instead it should at first try to collect a list of all object names, which need to be updated in a list, then eliminate duplicities in such a way that only the last occurrence of an object's name should remain in the list. Only then adjustDep of each object on the list should be called. Here is the code of the changed adjustDep procedure. It needs a helper procedure called prepareAdjust:

```
depObject'to objChanged
 let "objlist []
 foreach "ob dependlist
   [ask :ob [prepareAdjust]]
 foreach "ob :objlist
   [ask :ob [adjustDep]]
end
depObject'to prepareAdjust
 make "objlist lput myname butmember myname :objlist
 foreach "ob dependlist
   [ask :ob [prepareAdjust]]
end
```

After these changes in depObject, our geomShape object does not need to propagate changes in its adjustDep procedure, so it can be simplified like this:

```
geomShape'to adjustDep
 updateshape
end
```

Now you can move any of the red points and observe that p4 now moves quicker and smoothly.

This way a non-trivial example has helped us to improve the general object implementing dependencies. Now it is prepared to handle even more complex cases.

7. Conclusions and Final Considerations

In this paper we developed several classes in Imagine Logo, which help to implement projects having dependent objects. We have demonstrated these classes only in projects where the dependencies are spatial (position or shape), but the same approach can be used also for any other kinds of dependencies.

This paper has been written with the support of the European Community in the framework of the Socrates programme MINERVA 101301-CP-1-2002-1-HU-MINERVA-M.

8. References

Blaho A and Kalas I (2001), *Object Metaphor Helps Create Simple Logo Projects*, Proc. of Eurologo 2001, pp. 55 -65, ISBN 3-85403-156-4, Oesterreichische Computer Gesellschaft 2001.

GeomLand: http://www-iea.fmi.uni-sofia.bg/PGS/INDEX.HTM

Elica Logo: http://www.elica.net