Advanced Programming Classes in Imagine

Daniela Lehotska

Faculty of Mathematics, Physics and Informatics

Comenius University, Bratislava, Slovakia

lehotska@fmph.uniba.sk

Abstract

Future teachers of informatics at the Faculty of Mathematics, Physics and Informatics take Interactive programming and visual modelling course in the second year of their study. The aim of the course is to familiarize with the Imagine environment and its use for the development of educational applications, to learn about the principles of developing interactive, visual and open microworlds.

In the academic year 2004 - 2005 for the first time we offered students a continuation of this course – an optional seminar Programming classes in Imagine. The aim of this seminar is to apply more advanced programming methods for the development of medium-sized educational projects intended for children. In our paper we present the topics of the seminar. We give special attention to one class focused on working with programmable pictures. We show how to create a microworld for creating and editing the Escher tiles. We hope to illustrate several advanced Imagine Logo procedures and techniques (like higher-order procedures, modifying colours in shapes etc.), which the reader may productively apply in his/her own development of other microworlds.

Keywords

advanced programming in Imagine, tessellation, programmable pictures

1. Introduction

There are three streams of courses in the present concept of informatics teachers study at the Faculty of Mathematics, Physics and Informatics: essential mathematics, special informatics courses and didactics of informatics. Special informatics courses can be structured into three categories: programming category, applied subjects, and optional subjects. Programming category is based above all on the Delphi programming environment. In the second year of the study students take Interactive programming and visual modelling course that introduces the Imagine environment. *This programming in the Imagine environment builds a connection among the modern trends in programming (object, visual, parallel, event based programming), environment for the development of educational applications and a programming environment/language intended for teaching programming* (Blaho & Kalas 2004). In the academic year 2004 – 2005, for the first time, we offered students a continuation of the course – an optional seminar *Programming classes in Imagine*. The aim of the seminar is to apply more advanced programming methods for the development of medium-sized educational projects intended for children, to learn some tips and tricks in Imagine.

Two lessons have been allocated to this seminar per week. During these two lessons students together with the teacher developed one ore more small projects oriented to some topic. Students were asked to do homework every week, which was either completing a project from the seminar or creating a new small project using the same method as they learned during the class. With some projects we worked on for more seminars – we gradually improved them, added more functionality and settings, e.g. first we made a project for a simple player, later we created its net version etc.

2. Seminar topics

During the term we met students nine times. Corresponding nine topics are described briefly in the following table.

Table 1. Topics of the seminar, methods used to solve the problems and description of projects.

topic	instructions, problems	projects description
programmable shapes	spline, drawspline, outline map, generate	**screensaver** – curves created as a spline from positions of several random moving turtles **tessellation** – creating a tile by modifying a square and tiling
cutting and stamping pictures	putPicture, getImage, createSelection, define, choosers for opening and saving files	**jigsaw maker** – choosing a picture for a puzzle, cutting it into adjustable count of pieces, creating turtles (jigsaw pieces) with the shapes of these cuttings
working with audio and video	wave, midi melody and video objects, play, playWave	**playing icicles** – recording and playing melodies created by clicking on icicles; each icicle represents a certain tone
working with mouse cursor and keymenu	simulating big mouse cursor[2], setmc, keymenu	**playing icicles** – improving the previous version: adding stick as mouse cursor to strike on icicles and possibility to control the icicles with keyboard
changing the colour of a picture (not of a programma-ble shape)	shapecolor, tintcolor saving for web, saving project segments	**building blocks** – creating "constructions" using several types of building blocks which can be coloured by a colour from a palette saving the project for web, what are the restrictions in such case
working with net 1: sending text messages	net object and its properties, connect, user, connected?, send	simple **chat** project for sending (and receiving) text messages to all or specified net users
working with net 2: sending and receiving objects and	sending and receiving objects and instructions sendObject,	**building blocks** – net version of the project **net drawing** – every net user has its own turtle for drawing, drawing is send to all in the net **bomb game** – anybody in he net can create a bomb

[2] You can use your own picture (also a programmable one) for a mouse course, but the size of the picture can be at most 32x32. So if you want to use a bigger cursor, you have to simulate it by a turtle.

instructions	sendRun, onReceiveObject, runEnabled	that starts ticking from 10 to 1; while / when clicking on the bomb, the user sends it to random player; if the bomb reaches 0, it explodes
working with net 3	defining net object style during the run of the program, different functionality of server and client	In all previous net projects, a net object is created at the beginning and its style (server, client) is fixed. The project does not depend on the net object style - server and client do the same. In the **moodmeter project**, the net object is created during the run of the program and the style is defined according to the decision of the user. The client can choose its mood from five given smiles. Its selection is sent to the server, which processes the client's data, creates a graph and sends it to all clients.
graphs	graph representation	project for representation and visualization of graphs, simple graph algorithms

3. Tessellation project

The geometric meaning of the word tessellate is to cover the plane with a pattern in such a way as to leave no region uncovered (Schwartzman, 1994). Tessellations occur naturally in the world, and are frequently used in designs for works of art and architecture. The most famous art tessellations come from M. C. Escher who can be regarded as the 'Father' of modern tessellations. Tessellation can assist students in conceptualising infinity, learning about the different types of symmetry, and making observations about how colours and shapes affect perception.

Our tessellation project should enable creating a tessellating tile by square modification (deformation) and subsequently doing tessellation with it.

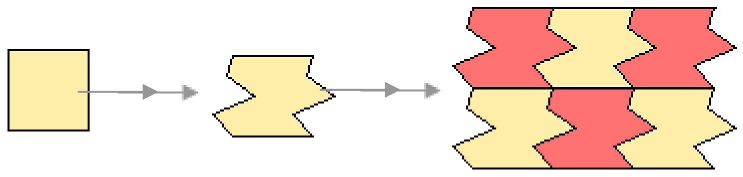

Figure 1. Square → tile → tessellation.

Let us do a simple analysis of a project. A tile will be represented by a turtle with a shape of a square at the beginning. Several small black points (turtles) will be placed on one side of the square. It will be possible to drag these points, except vertices, and deform the square in this way. The deformation will be applied in a certain modified way to another side of the square as well.

There are many possibilities of modification, e.g. translation, rotation around a vertex, reflection through the diagonal, etc. First of all we realize the modification, where the left vertical side is translated to the right vertical side. Horizontal sides remain without change.

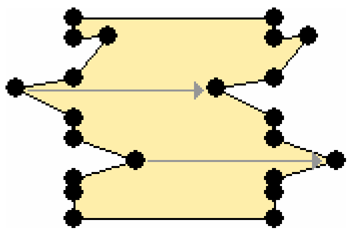

Figure 2. Translation of the points from the left to the right vertical side.

3.1. Creating a tile

First, we create a turtle-tile at the position `[0 0]`, with a shape of a yellow square sized `100` and name it `tile`:

```
new "turtle [name tile pos [0 0] fillcolour yellow
             shape [polygon [repeat 4 [fd 100 rt 90]]]]
```

It would be nice to enable resizing of the square. To do this we put the slider named `sWidth` with the range from `2` to `10` on the page. The width of the tile can be `10` times the value of the slider. The width of the tile will be used in many procedures later, so we set up a variable `tileWidth` that will be updated when the value of the slider is changed. We define a `tile`'s procedure `myShape0` for changing the shape of the square according to the actual value of `tileWidth` and we define `onChange` event for the `sWidth` slider like this: `make "tileWidth 10*value tile'myShape0.`

```
to myShape0
  setShape ![polygon [repeat 4 [fd :tileWidth rt 90]]]
end
```

We create a new class for the small dragable points: with common name prefix `point`, switched on `autodrag` property, shape of a point of size `8` and pen `up`:

```
newClass "turtle "point [common'namePrefix point autoDrag true
                        shape [point 8] pen pu]
```

Now we generate some points placed on the left vertical side of the square with the `10` pixels spacing. If the square width is `100`, then we must create `11` points (in order to include both segment vertices). Vertices should be fixed, so we switch off their `autodrag` property. As the count of the points depends on the slider's value, we define a procedure for generating them – `newTile`.

```
to newTile
  eraseObject allOf "point
  repeat (sWidth'value + 1)
   [new "point [xCor 0 yCor (10*(repc-1))]]
    ask se first allOf "point last allOf "point [setAutodrag "false]
end
```

Now it's time to make it possible to modify the tile's shape while dragging the points. We add `onDrag` event to the points with the definition `tile'setShape1`, where `setShape1` is a tile's procedure for changing shape using the translation modification (see figure 2). The shape of a tile is created by connecting corresponding points with line segments (see figure 3).

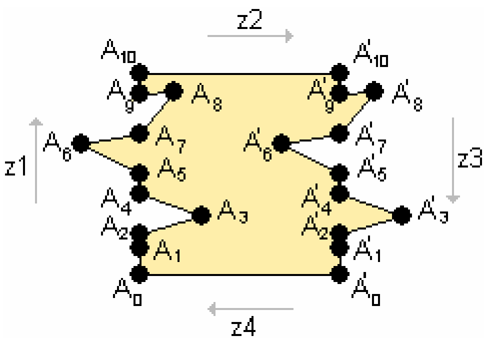

Figure 3. Creating tile's shape out of points

Let **v** = `[:tileWidth 0]` be a vector of translation. Let A_0, .., A_{10} (see figure 3) be the positions of point-turtles on the left side of the square. We collect them in the `z1` list. By translation of points A_0, .., A_{10} (given by vector **v**) we get points on the right side of the square $A'_i = A_i + \mathbf{v}$. They are stored in `z3` list. The outline of the tile is made by connecting the points A_0, A1, .., A_{10}, A'_{10}, A'_9, .. , A'_0 (in this order!) with line segments. We use the `outline` command to define a proper shape with a list created as a connection of `z1` list and reversed `z3` list as a parameter. If we use – as an extension – the `polygon` command, we get not just an outline but a filled tile.

```
to myShape1
  ; left vertical side (positions of point-turtles)
  let "z1 map [ask :% [pos]] allOf "point
  let "translation ![:tileWidth 0]
  ; right vertical side is created by translation [:tileWidth 0]
    from the left one
  let "z3 map [:% + :translation] reverse :z1
  ; merging z1 and z3 lists
  let "shape se :z1 :z3
  setShape ![polygon [outline :shape]]
end
```

Now, if we move the slider, the square shape is changed, but the points remain at the same positions. By changing the size of the tile, the shape of the tile must be changed and all the point turtles must be regenerated. We define these two actions in the `newTile` procedure and then change the `onChange` event of the slider to: `newTile`.

```
to newTile
  make "tileWidth 10*sWidth'value
  eraseObject allOf "point
  tile'myShape0
  repeat (sWidth'value + 1)
    [new "point [xCor 0 yCor (10*(repc-1)) onDrag [tile'myShape1]]]
  ask se first allOf "point last allOf "point [setAutodrag "false]
end
```

Notice that we also add setting the `onDrag` event of the point-turtles to `tile'myShape1`.

The shape of the tile is angular. What about to add a possibility of creating a rounded shape as well? We create two buttons named `bAngular` and `bRound` – both switch with the same group number (e.g. `1`) and switched off `All Buttons May Be Off` property, possibly we draw some appropriate icons for them. We have to change the `myShape1` procedure. To create a rounded shape, we use the `spline` operation. `Spline` gets a list of points as input and outputs a modified list of points – several additional points are inserted into the list so that the whole curve is smooth. Important! The list of points on the left and right side must be splined independently.

```
to myShape1
  ...
  let "shape ifElse bAngular'down
    [se :z1 :z3]
    [se spline :z1 spline :z3]
  setShape ![polygon [outline :shape]
end
```

The shape of the tile should change by pushing the `bAngular` and `bRound` switches. So we add `tile'myShape1` command to their `onPush` event.

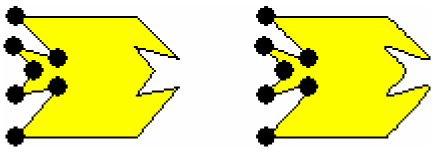
Figure 4. Example of an angular and a rounded tile.

3.2. Tessellation

The tile is ready, let's do the tessellation. There are two possibilities of doing tessellation: automatically (we define a procedure for putting tiles properly to each other) or manually (we create many copies of the tile and let the user to put them on the plane). We show the automatic way.

We add another page to the project – this will be the place for tessellation. The most simple solution for creating tessellation would be using the `putPicture` command with the shape of the `tile-turtle` from `page1` as the first parameter and the `style` property set to the value `tile: putPicture page1'tile'shape [style tile].`

If we try that, we run into some limitations of our solution. First of them is that all tiles are of black colour due to the fact that `putPicture` command gets from the turtle only the drawing list without any fill colour information. Another limitation is that tiles do not fit into each other, see figure 5. The reason is that `putPicture` stamps the rectangle in which the shape is inscribed and shifts by the length of that rectangle, instead of shifting by the original width of the sqare that is the basis of the tile.

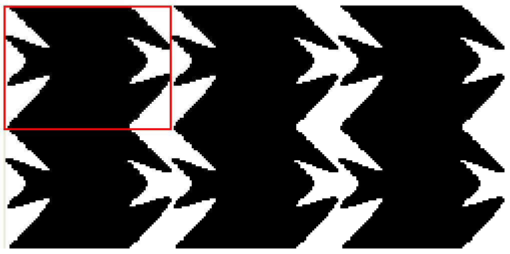
Figure 5. Example of an incorrectly made tessellation using putPicture command.

So we cannot use the `putPicture` command, we have to define the tessellation by ourselves. We will use a turtle that moves by the step `:tileWidth` on `page2` and stamps the `tile`'s shape. To make the calculation easier will set the origin of `page2` so that it is in the lower left corner (`X 0`, `Y 499` by default size settings). Here is how to create the stamping turtle:

```
new "turtle [name stamp pen pu fillColour yellow pos [0 0]
             rangeStyle window shown false]
```

By transition from `page1` to `page2`, the `stamp`-turtle gets the `tile`'s shape and then realises the tessellation. This is defined in the `prepare` procedure which is called when the `page2'onShowPage` event comes up.

```
to prepare
  setShape page1'tile'getShape
  tessellate1
end
```

The tessellation is realised as a two-dimensional cycle (through rows and columns). While stamping the tile, we switch between two colours, yellow and red. To change the colour of the `stamp`'s shape, we have to change its `fillColour` and then reset its shape.

```
to tessellate1
  cs
  let "columns 1 + div (first page1'size) :tileWidth
  repeat 1 + div (last page1'size) :tileWidth
    [repeat :columns
     [;colours is a global variable with starting value [yellow red]
       setFc first :colours
       setShape shape
       stamp
       make "colours reverse :colours   ;changing the colours order
       setXCor xCor + :tileWidth]
     if 0 = mod :columns 2 [make "colours reverse :colours]
     setXY 0 yCor + :tileWidth]
end
```

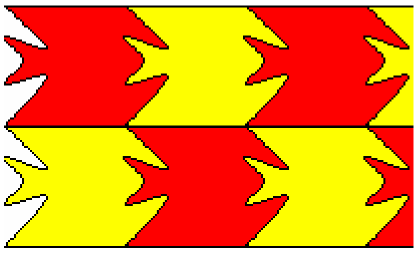

Figure 6. Example of a correctly made tessellation using own procedure.

To finish the first version of the project we link the two pages using buttons.

3.3. Other types of tiles

In this part another three possibilities of tile modification are presented (see figure 7).

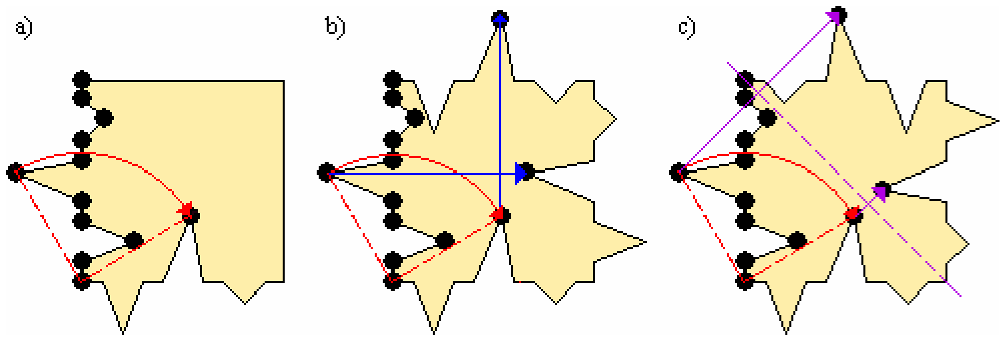

Figure 7. Three tile modifications: a) rotation, b) rotation and translation c) rotation and reflection.

To realise these three modifications, we define three procedures of tile-turtle: `myShape2`, `myShape3` and `myShape4`. We use local variables `z1`, `z2`, `z3` and `z4` to denote the lists of points of the left vertical, upper horizontal, right vertical, and bottom horizontal side (in that order). Procedures are commented in the code. Notice that each point transformation is made in parallel with all points by mapping the corresponding procedure to the whole list of points.

a) Rotation (see Figure 7a)

```
to myShape2
  ; left vertical side (positions of point-turtles)
  let "z1 map [ask :% [pos]] allOf "point
  ; bottom horizontal side is created from the left vertical side by
    rotation around [0 0] about 90°
  let "z4 map [rotatePoint [0 0] 90 :%] reverse :z1
  ; upper right corner
  let "b ![[:tileWidth :tileWidth]]
  ; merging z1 list, b and z4 lists
   let "shape ifElse bAngular'down
    [(se :z1 :b :z4)]
    [(se spline :z1 :b spline :z4)]
  setShape ![polygon [outline :shape]]
end
```

In this procedure we use our `rotatePoint` procedure, which outputs the coordinates of a point rotated around a given centre by a given angle.

```
to rotatePoint :centre :angle :point
  op :centre + rotate :angle :point - :angle
end
```

b) Rotation and translation (see Figure 7b)

```
to myShape3
  let "z1 map [ask :% [pos]] allOf "point
  let "translation ![:tileWidth 0]
  ; right vertical side is created by translation [:tileWidth 0]
    from the left one
  let "z3 map [:% + :translation] reverse :z1
  ; bottom horizontal side is created from the left vertical side by
    rotation around [0 0] about 90°
  let "z4 map [rotatePoint [0 0] 90 :%] reverse :z1
  let "translation ![0 :tileWidth]
  ; upper horizontal side is created by translation [0 :tileWidth]
    from the bottom one
  let "z2 map [:% + :translation] reverse :z4
  ; merging all the list to one
  let "shape ifElse bAngular'down
    [(se :z1 :z2 :z3 :z4)]
    [(se spline :z1 spline :z2 spline :z3 spline :z4)]
  setShape ![polygon [outline :shape]]
end
```

c) Rotation and reflection (see Figure 7c)

```
to myShape4
  let "z1 map [ask :% [pos]] allOf "point
  ; bottom horizontal side is created from the left vertical side by
    rotation around [0 0] about 90°
  let "z4 map [rotatePoint [0 0] 90 :%] reverse :z1
  let "b ![:tileWidth :tileWidth]
  ; upper horizontal is created by reflection (width a diagonal
    as the symmetry line) from the left vertical side
  let "z2 map [:b - reverse :%] reverse :z1
  ; right vertical side is created from the bottom horizontal side
    by the same reflection
  let "z3 map [:b - reverse :%] reverse :z4
  ; merging all the list to one
  let "shape ifElse bAngular'down
    [(se :z1 :z2 :z3 :z4)]
```

```
      [(se spline :z1 spline :z2 spline :z3 spline :z4)]
    setShape ![polygon [outline :shape]]
  end
```

We add four buttons for setting the type of the tile modification – these are switches with the group number set to `2` and `All Buttons May Be Off` property switched off. When any of these buttons is pushed, we set the global[3] variable `tileType` to the number which determines the type of tile modification, and then call general procedure `myShape` of `tile`-turtle. This procedure decides – based on the `tileType` value – whether `myShape1`, `myShape2`, `myShape3` or `myShape4` should be run. So the `onPush` event for the first button looks like: `make "tileType 1 tile'myShape`. The `myShape` procedure can be defined in this short way.

```
  to myShape
    run word "myShape :tileType
  end
```

Now we have to rewrite the `onDrag` event of the `point`-turtles and the `onPush` events of `bAngular` a `bRound` buttons to: `tile'myShape`.

Also the way of tessellation has to be changed because the `tessellation1` procedure is not appropriate for all types of tiles. E.g. the simple way of tessellation with the tile created by rotation (`myShape2`) is to stamp not 1, but 4 tiles in one step – the original one and its `90`, `180` and `270` degrees rotations.

```
  to tessellate2
    cs
    repeat 1 + div (last page1'size) 2*:tileWidth
      [repeat 2 + div (first page1'size) 2*:tileWidth
        [repeat 4
          [setFc first :colours
           setShape shape
           stamp
           rt 90
           make "colours reverse :colours]
         setXCor xCor + 2*:tileWidth]
       setXY 0 yCor + 2*:tileWidth]
  end
```

We define also `tessellation3` and `tessellation4` procedures for two remaining tile types. To run the proper version, we rewrite the `prepare` procedure on `page2`.

```
  to prepare
    setShape page1'tile'getShape
    run word "tessellate :tileType
  end
```

3.4. Further suggestions

In this part we present suggestions for further development and improvement of the project. They are structured in three groups.

1. Creating the tile.

- We can add more types of tiles. There are many combinations of congruent plane transformations that can be applied to one or more sides of the square.

[3] global because we need it also in page2 for determining the type of tessellation

- We have used square as a basic shape, but it is also possible to start from some other plane figures, such as equilateral triangle, regular hexagon, parallelogram, diamond etc.
- The dragable points were placed on the side evenly with the spacing of `10` pixels. The spacing (or count) of the points could be defined by a slider. Potentially, it could be possible to drag an arbitrary point of the square side.

2. **Automatic tessellation.** We used two strictly defined colours (yellow and red) for the tessellation. One way of changing it is to fill the tiles with random colours. But it would be nice to let the choice of the colours on the user. There are several possibilities how to realize it.
   - Before doing the tessellation the user chooses two colours from a palette.
   - We do the tessellation with blank filling (white in fact) and after doing the tessellation we offer the user a colour palette and a fill tool to fill the tiles as he/she wants.
3. **Manually tessellation** has already been mentioned. To do the tessellation manually, the user should have tools for rotating (by `90` will do in our project) and reflecting tiles, setting their colours and of course dragging them. To help the user, the tiles could stick to the closest grid position after releasing the mouse button.

It's apparent that project containing all these possibilities, functions and settings cannot be developed in one or two lessons. Probably we can elaborate on it for the whole term, but then the students will not meet many other interesting topics and ideas. So we can let these further suggestions as a motivation.

You can try the project at http://user.edi.fmph.uniba.sk/lehotska/tessellation.html or download the imp-file from http://user.edi.fmph.uniba.sk/lehotska/tessellation.zip.

References

Blaho A and Kalas I (2004), *Príprava novej koncepcie štúdia učiteľstva informatiky na FMFI UK* (in Slovak), DidInfo 2004, Univerzita Mateja Bela, Fakulta prírodných viet, Banská Bystrica 2004, ISBN 80-8055-908-2, pp. 10–13.

Schwartzman S (1994), *The Words of Mathematics – An Etymological Dictionary of Mathematical Terms Used in English*, 1994, Mathematical Association of America.

Annal D (2003 - 2005), *Tesselations – Escher and how to make your own.* Retrieved on April 20, 2005 from http://www.tessellations.org/ .

Tesselate! The Shodor Education Foundation, Inc. Retrieved on April 20, 2005 from http://www.shodor.org/interactivate/activities/tessellate/what.html