Australian National University  **College of Eng. and Comp. Science**

Research School of Computer Science

COMP6700 • COMP2140
Assignment 2

Morphing Shapes

Due 11:59pm EST on Monday, May 23, 2016

Total mark: 15

Abstract

The second assignment requires you to complete an unfinished program which uses the JavaFX API to create a simple graphic program. It will include effects of transition and animation (when the graphics scene changes with time), and control elements (menu and buttons) which allow a user to select a shape object (shown in the scene), change its properties, morph into a different standard shape, save and clear the scene. Apart from use of JavaFX's API, the tasks involve creation and implementation of simple computational geometry algorithms. The use of λ-expressions and stream processing is encouraged but not necessary.

This assignment covers most of the topics we discuss in Sections **A**, **F** and **R** of the official lecture plan (see COMP6700/2140 Lectures) The completion of the project requires a substantial self-study of how to program a rich graphic application using JavaFX API (references are provided).

Introduction

This program will create a simple graphical editor which allows one to draw arbitrary (*free-hand*) 2D shapes by using click-drag-release mouse (or track pad) operations. Originally, the program can only draw at most two shapes, but this is an artificial restriction which you will have to remove. The program can also execute a *morphism* — a continuous transition of one shape into another. You will utilise this feature to program various kind of morphisms to transform an arbitrary shape into one of the standard, "exact" geometric shape (polygon, rectangle and ellipse). A graphical user interface (*GUI*) application is usually operated by a set of *control elements* — menu and menu items, buttons, check boxes *etc*. At the moment, the program is devoid of them except for rudimentary keyboard controls which allow to trigger a morphism between two shapes (keys `meta-N` and `meta-M`). Your other task is to add a simple menu of controls which can be used to select a shape element, enact its transition into a chosen shape, and also perform one of the standard file operations of saving and opening. The operation of saving will convert each displayed shape into a simple string and write all such strings in a plain text file (use the `.txt` extension!), and the operation of opening will perform the reverse — read string representations and create *Path* objects and display them in the application window.

A non-graphical part of the assignment is to program the underlying representation of the standard shape into which a "hand-drawn" shape will morph. Even though *JavaFX* does have API classes to represent each of the "standard shapes" (*Circle, Ellipse, Rectangle, Polygon*) in order to perform an animated morphism into one of these shapes from an arbitrary shape, the JavaFX animation API demands that one needs to use a (filled) *Path* object rather than these standard shape classes. Therefore, when you perform a morphism of an arbitrary shape (which will itself be represented as a *Path* object) into a standard shape, you will have to calculate the parameters of the target shape as a Path object.

Background: JavaFX: controls, layout, css-styling, graphics and effects

JavaFX is a set of API classes to create a multi-windowed application which can display a set of graphically rich control elements, shapes and complex dynamics of changing and moving. We discuss the basic aspects of graphics programming, the use of widgets to control graphical user interface, and the creation and manipulation of rich graphics effects, such as spatial transformations (translation, rotation and scaling) and continuous transitions (animation), in an application in the section **R** of lectures, which slides you should study. Other references are provided in the References.

The JavaFX API (plus SDK tools, plus either the scene layout editor *SceneBuilder* or a suitable IDE plugin) can be used to create:

- an application layout as a collection of containers and control elements
- a programmed response to signal which a user can generated using these control elements
- enhanced visual appearance of the user interface with css-files
- various "dynamic" effects when the graphical shapes and other components change their properties (colours, size, form, position *etc*) continuously in the course of the running application
- 3D graphics with basic (but not too sophisticated like in, eg, *OpenGL*) object manipulation

Different parts (`javafx` sub-packages) of the API are used in the above programming aspects; you should familiarise yourself with the JavaFX API structure, and study basic examples of how they are used from the JavaFX Tutorial.

What is already done?

The starting point for your work is a program consisting of the following Java source files and an `fxml`-file:

1. `shapechanger.ShapeChanger` – the main class which is also `javafx.applicaton.Application` class (every JavaFX application must contain one); it creates a scene and defines several callbacks (event handlers) which make the program respond to mouse events (`press`, `drag`, `release`) and keyboard events (`meta-N`, `meta-M`, `meta-Q`). No other control elements (*widgets*) are currently programmed.
2. `shapechanger.Controller` — the automatically generated (currently empty) class; you may ignore and leave it unchanged unless you decide to use `fxml`-based layout (in which case all the control elements will be declared in the `shapechanger.view.fxml` layout file, normally generated by the tools like *SceneBuilder*). **It is not the assignment requirement to use `fxml`/`css`-based approach** to create the user interface to create the user interface layout, but you may follow such approach if you so wish.
3. `shapechanger.Morph` — the Model part of the program (in the *Model-View-Controller* architecture sense); it allows the client program (the `shapechanger.ShapeChanger` class) to create an internal (meaning, *non-graphical*) representation of shapes which are processed and display by the View component of the program (part of the *ShapeChanger* class). A model representation of a *Path* object is *Morph*, which defines several methods needed to perform Path transformation needed for animated morphing. You will add more methods which perform various other morphing effects.
4. `shapechanger.Point` a small helper class which is used as a replacement of `javafx.geometry.Point2D` (it's a slimmer type, but it also implements a design decision to make Model independent of View).

This is how a morphing effect executed by *ShapeChanger* looks like currently:

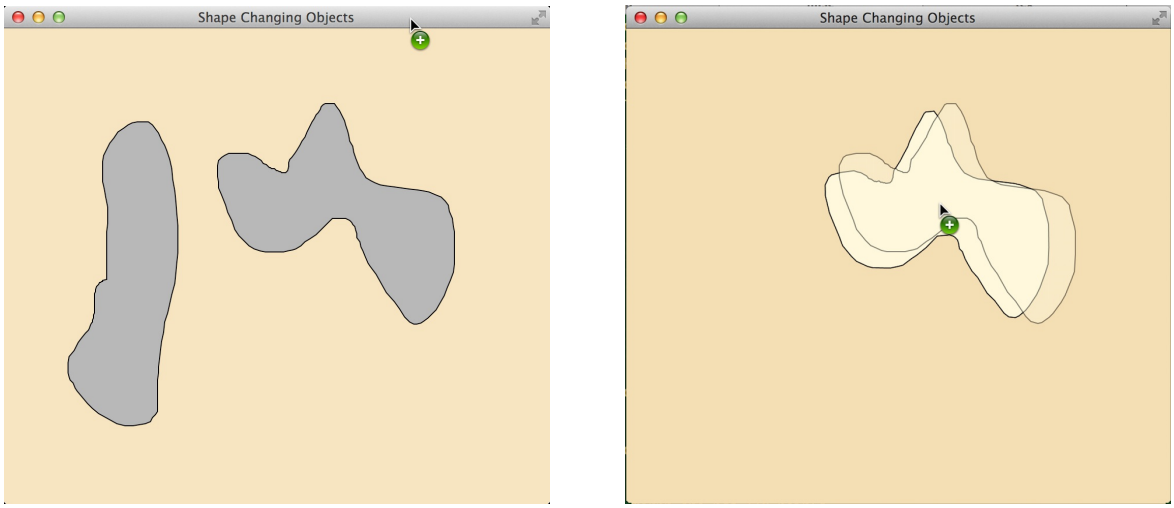

Figure 1: **On the left:** two paths drawn by mouse dragging, **on the right:** during a transition when the left path is morphing into the right one (change of colour is for pure decoration).

To facilitate the code use (in an IDE-neutral way), I provide a `Makefile` to simplify the compilation, execution, and generation of documentation (using `javadoc` tool). The `Makefile` use is explained in the `README` file. You will not need the `Makefile` if you decide (and you should!) to use an IDE, but, please, **do not change the package declarations** (for the existing classes) when you extend the program; if you define new classes which need to be placed in a separate package, make this package extend the `shapechanger` name, *eg*, `shapechanger.view` or similar.

Tasks

You are asked to implement the following effects:

1. Extend the main window scene by creating a `Manubar` with the following `Menus`:

   File with menu items `Open`, `Save` and `Quit`

   Edit with a menu item `Select`

   Morph with menu items `Triangle`, `Ellipse`, `Rectangle` and `Polygon` to initiate a morphism of a selected shape object into a "standard" shape; for `Polygon`, the program will open a small dialog window with a single field asking the user to type in the number of the sides that the polygon has. The orientation of the target shapes will be default: Triangle will have a horizontal base and a vertical height; Rectangle will coincide with the path *bounding box*, Ellipse will be the ellipse with the same bounding box as the path itself, Polygon will be an all-sides-equal polygon which is symmetric with respect to a vertical axis passing through the centre of the arbitrary shape.

2. Program the application response to the events generated by the menu item controls described above:

   File for Save, the program must convert all drawn shapes (only the paths describing their border since we assume that the path border is stored as fields and all its other properties have default values); if the scene has no drawn shapes, this operation should warn the user that there is nothing to save instead of creating an empty file; the file name selection should be done using the JavaFX's `javafx.stage.FileChooser` class; for Open, the program must query the user (again, by using `javafx.stage.FileChooser`) about a file with saved path strings, read the chosen file, create and display shapes (you should assume that *all* files which are opened from *ShapeShanger* will be path description files — in other words, do not expect to get bad formatting exceptions which you need to catch to avoid crashing the application); and for Quit, make the program terminate (asking if the user needs to save an open non-empty file).

Edit for `Select` menu item, the event handler should switch into *selection mode*. The mode will make the application react differently to mouse events: for the *mouse move event*, when the mouse (or track pad) cursor hovers over a path object, the visual presentation of this object should indicate that it **selectable** (make the colour of the shape border noticeably different, or make the border flicker, or something...); when the *mouse press event* occurs on a selectable shape, this shape becomes **selected** and can later be morphed when the **Morph** menu is used (once again, the visual representation of the shape should change to indicate that it has been *selected*). Note that the region where a shape becomes selectable will be defined to be the bounding box of that shape. When a mouse move event enters a region which is contained in the bounding boxes of two or more shapes, make the shape whose bounding box has been entered first the "selectable shape" (don't worry about cases where two different shapes might have exactly the same bounding box).

Morph when one of the target shapes is selected and the program has one of its shapes marked as `selected` (the program shall have at most one shape object selected for morphism, not more), that shape will undergo a morphism transition into the chosen "regular" shape, after which it will not be `selected` so the program can select a new shape for morphing.

3. (Advanced) Consider replacing the simple target shapes of `Triangle`, `Ellipse`, `Rectangle` and `Polygon`, which are horizontally oriented, into more "authentic" shapes, whose orientation is tilted; an example of morphing a path to horizontal and tiled rectangle is depicted below:

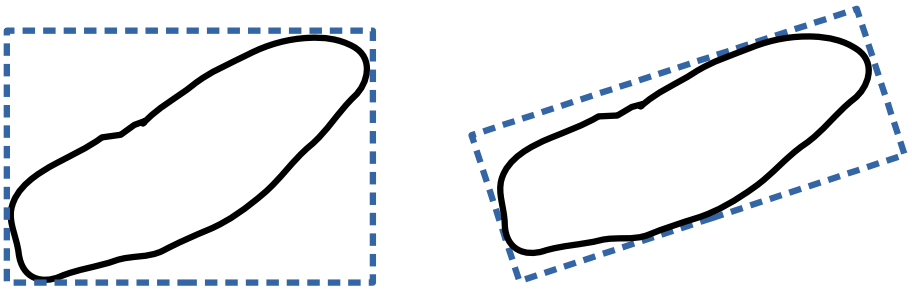

Figure 2: "Naïve" (**left**) and "Advanced" (**right**) morphisms of a shape to rectangle.

The target shapes to be considered are ellipse and rectangle only, their cases are essentially similar — by determining a tilted rectangle, the ellipse is defined by the tilt angle and its semi-axes are halved width and height of the rectangle. The algorithm which determines the tilt is based on calculation two points on the original path which are separated by the maximal distance; a line passing through these two points will determine the tilt angle.

The Marking Guide

The marks will be awarded in accordance with the following scheme. (Note that the allocated mark distribution is a guide only and that the quality of the software that you write may impact on the marks received; scrappy or poorly documented software will lose marks.)

2 points for adding the control menu (as described above) *without programming the callbacks*

5 points for programming the callbacks needed to implement the file opening and saving features, including converting path shape objects into string representations and writing them to a file, and all the reverse file-opening operations

2 points for programming the callback `Edit->Select` and extending the corresponding parts of the program to mark a shape as `selected`

4 points for programming the callbacks to implement morphism of a selected shape to a chosen target shape (in the naïve version)

2 points for implementing the advanced morphism (as described above)

Submission: *GitLab*

The project work and submission will be done via the use of a *GitLab* repository comp6700-2016 which you have to fork thus creating your own repository with the `URL`

```
https://gitlab.cecs.anu.edu.au/u0123456/comp6700-2016
```

(`u0123456` should be replaced by your University ID), which you will clone, modify by adding and modifying code to fulfil the assignment requirements, commit those changes and, finally push to the server repository (the one you've forked) as your submission. The detailed instructions of how to use *Git* and the project source code management and submission are explained in detail in the Lab 6. The Git repository (and submission) will also include your homework 7 and 8.

Important When you fork the master repository to create your own copy, you need to add the course lecturers and the tutor as *reporters* of your project. We need a read access to your repository for marking purposes. It is not appropriate to commit any data (especial private and/or sensitive) which are not related to the course homework and assignment.

Let me know if there are problems with submission (preferably before the deadline).

Last Remarks

If you notice an error (typo, poor language, ambiguity or contradiction) in this document let me know as soon as possible, so I can correct it for everyone's benefit.

Send an email or come and see the lecturer or your tutor for consultation if you need to discuss the assignment and/or your work.

References

[1] JavaFX API

[2] JavaFX Tutorial

Alexei Khorev, Henry Gardner

`version 0.8`, April 26, 2016
`version 1.0`, April 29, 2016
(title change, minor language improvements)