IBRAHIM, Mohamad Hassan

IGR201b _ Qt TP 2

Questions Answered:

Q: The display list will contain objects of **different types** (lines, rectangles, ellipses). What is the typical solution that is used in Object-Oriented Programming for solving such a problem?

A: Polymorphism which uses a base class called an interface and lets other child classes inherit its methods to use differently.

File Structure:

main.cpp: creates the application and initializes MainWindow

mainwindow.h / mainwindow.cpp: builds the program tools (menu bar, toolbar, and places the drawing area) and handles the close event

- Menu bar contains the color, width, style and shape options
- Tool bar contains the edit mode toggle switch and the save button

Canvas.h / Canvas.cpp: The drawing area that is initialized by mainwindow

- Handles mouse events (press, move, and release)
- Handles the drawing and editing of shapes
- Stores the display list in QList of pointers called shapes that is handled by paintEvent
- Implements the setters and getters for pen menus, tools, and edit mode
- Implements the image export (save) that saves the image in saved_pictures directory

Shape.h: Polymorphism Class used to handle different shapes

- Base class called Shape that have methods to paint and edit the shapes
- Child classes LineShape, RectShape, and EllipseShape that inherit and implement methods of the base class Shape

Steps Summary:

Step 1: Create a new project

- Created the project file in qt creator latest version and added the Canvas class header and source file
- Initialize the Canvas in the MainWindow class

Step 2: Drawing a line interactively

- Used the mouse events to detect mouse pressing, moving, and releasing to draw a single line
- Use drawLine method that is part of QPainter
- Line starts from startPoint and follows currentPoint until the release of the mouse (mouse press sets currentPoint = startPoint, mouse move updates currentPoint, and mouse release fixes currentPoint in its place)

Step 3: Specifying graphical attributes

- Added Menus from the mainwindow class using QAction and QActionGroups
- For more flexibility in future work and to make it easier to implement other menu options, I created a single method called createPenMenu and multiple methods called addAction…. (color, width, or style) and the createPenMenu calls these methods to add them to the menu, therefore in the future I can create a new Action (Menu option) and just call it from the createPenMenu.

Step 4: Drawing other geometric shapes

- Added a shape menu that is separate from the Pen menu (although I could have created a main menu and added all pen and shape menu to it)
- This shape menu allows the user to choose one of the 3 shapes to draw (line, rect, ellipse)
- Added a switch block in the paintEvent so that I call different methods to draw depending on the chosen shape / tool.
- Line uses drawLine, Rectangle uses drawRect which in turn uses a normalized QRectF, Ellipse uses drawEllipse which also uses a normalized QRectF

Step 5: Drawing multiple shapes

- This step introduces Polymorphism as asked (Shape.h is created here with the base and child classes), but here they only implemented paint method.
- To draw multiple shapes on the screen, I also introduced the QList<Shape*> that is a list of pointers of type shape each pointer points to a shape that the user has

drawn (if the user draws a rectangle we will append a pointer of type Shape that is pointing to RectShape)

- Changed paintEvent to paint the shapes in the shapes QList
- Optionally, I added a destructor that deletes all created objects when terminating to not have memory leaks

Step 6: Editing the shapes

- Added Edit mode toggle so that when it is OFF (default) the user can draw and when it is ON the user can select shapes to edit
- Implemented 4 new methods in Shape.h that are hitTest, moveBy, resizeTo, paintSelection and another helper method called handleRectAt all work while the edit mode is ON
    - hitTest: tells if the click on the edge (return the number of the edge), on the body (returns -1), or on a blank space with no shapes (returns -2 for no hit)
    - moveBy: translates the shape by delta = currentMousePosition – lastMousePosition using the built in translate method
    - resizeTo: after detecting the edge clicked, the user can resize the shape based on the current mouse point (clicked edge follows the mouse)
    - paintSelection: when a shape is selected, this method draws a dashed selection outline
    - handleRectAt: draws the little squares over the edges of the shape
- **Note for hitTest:**
    - For lines, returns 0 or 1 if we hit an edge box, else it creates a small ‘X’ shape where the user clicks, if one of the segments of this ‘X’ hits the line we have an intersection and it returns -1
    - For rects and ellipse, returns 0 to 3 if we hit one of the edges, else we create another rect or ellipse inside the original one, if the click is between the inner shape and the original shape it returns 4 and if it is inside the inner shape it returns -1
    - -1 means we can move the shape
    - 0-4 means we can resize the shape
    - hitTest is called by the mouse press event which updates the activeHandle (returned value of the hitTest)
- **Note for resizeTo:**
    - For lines, when we click a handle and move it we will just update line.setP0 or P1 depending on the handle we clicked

- For rects and ellipse, if the handle clicked is between 0 and 3 it means we found an edge and we can edit it easily by updating the position of this handle, but if the click returned 4 we need to find the nearest edge to the click and then we assume that we clicked this edge and we can resize.

Step 7: Ask for confirmation before exiting the program

- Introduced a dirty flag, always false at the beginning, but every time the user touches the screen this flag becomes true, and the app now requires to be saved by clicking the save button
- If the user tries to exit without saving, the program will show 3 options (save, discard, cancel)
  - save: will save the work as if he clicked the save button and then quits
  - discard: will quit the program without saving, and the work is lost
  - cancel: will cancel the exiting process and keep the user in
- The drawings are saved in a JPG format in saved_pictures directory created the first time a user tries to save a drawing

NOTE: The program was tested and everything was working fine, tested it on a windows 11 machine and on a Kali Linux Virtual Machine

NOTE: Here is the link for the GitHub repository:

https://github.com/M-Hasan-Ibrahim/Qt_Paint

IBRAHIM, Mohamad Hassan

mohamad.ibrahim@telecom-paris.fr