THE HONG KONG
POLYTECHNIC UNIVERSITY
香港理工大學

Computer Graphics Toolkit Report

OF

COMP4422 COMPUTER GRAPHICS INDIVIDUAL ASSIGNMENT

FOR THE

PROFESSOR JANE and TEACHING ASSISTANCES

NAME: CHEN YI PU
STUDENT ID: 19084858D

Ref. No.: REP001
Version: 0.1.0

October 2023

Table of Content

1 System Design

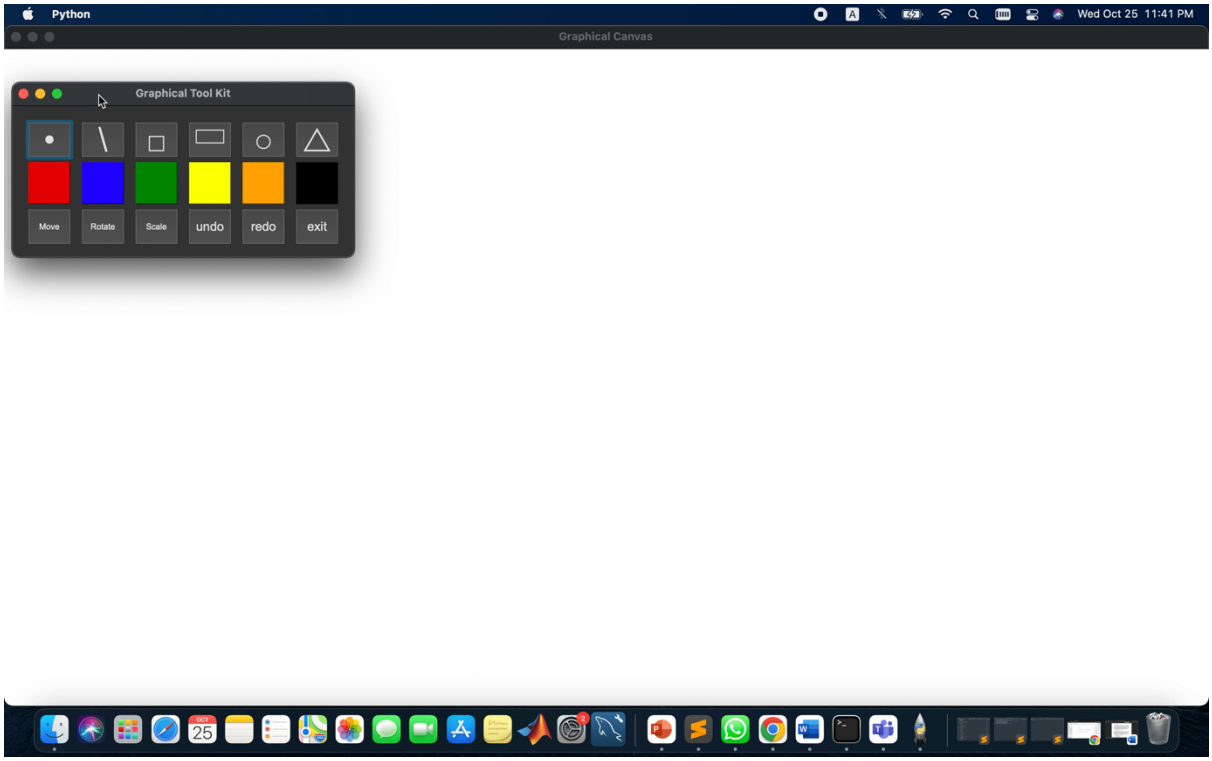
The Computer Graphics (CG) Toolkit is a program simulating off-the-shelf CG drawing software in terms of 2D graphics creation and manipulation. The simulation is accomplished by providing interactions through a user interface with built-in functions for creating objects, CG primitives, performing translation, rotation, scaling, and additional editing features. With minimal setup and installation of Python and packages, the program satisfies platform independent and even limited customization on program setting parameters. As a result, in terms of development, Python was chosen as the main programming language. The program utilizes the PyQt library and implemented in an Object-Oriented approach enabling any further extension and implementation with minimal work.

1.1 System Architecture

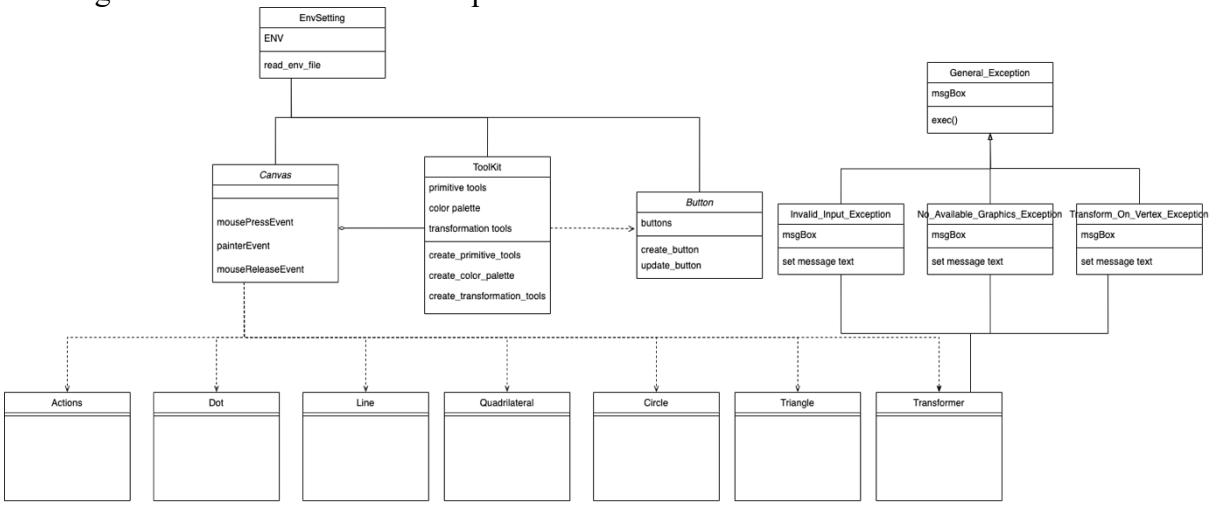
From a system architecture point of view, it is obvious that the Toolkit generates or manipulates graphics based on the communication and interactions between the tools window and the canvas window. These, indeed, are the two main Classes lie as basis of the system. The execution starts by initializing a blank canvas and a window of buttons, the tools window. The buttons are created along with the initialization of the tools window by creating a Buttons object and calling corresponding functions. Each button represents a function or action for basic CG operations. Available functions include:

- Drawing Primitives: Dot, Line, Square, Rectangle, Circle, and Triangle
- Coloring
- Translation to user input x and y variance
- Rotation to user input angle
- Scaling to user input number
- Undo and redo actions
- Exit program

In order to replicate the full control in other CG applications, the canvas constantly listen on user actions, named events, including mouse click, mouse release, and mouse movement. Such implementation enables users to draw CG primitives with customized length or size, based on clicking, dragging, and dropping, not as input but in an intuitive manner.



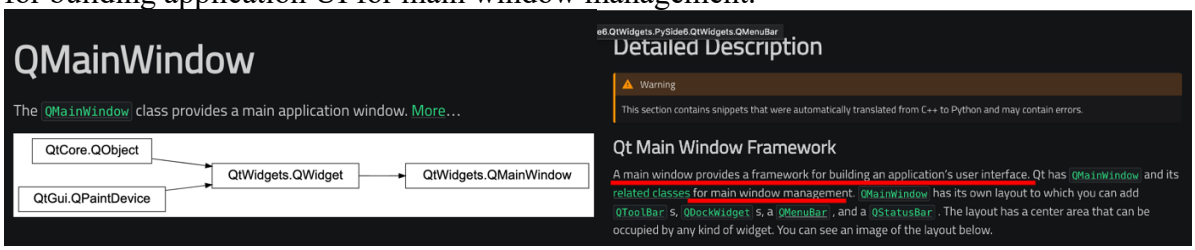
The logics of each function are implemented in server Packages. There are two server Packages, namely action_service and general_service. The former defines the usage and operations of each button as well as interprets and stores the events passed by the canvas window. The general_service provides services commonly shared by different classes. Together, the five aforementioned components formed the core of the Toolkit, jointly working with other small Classes to present the final result.



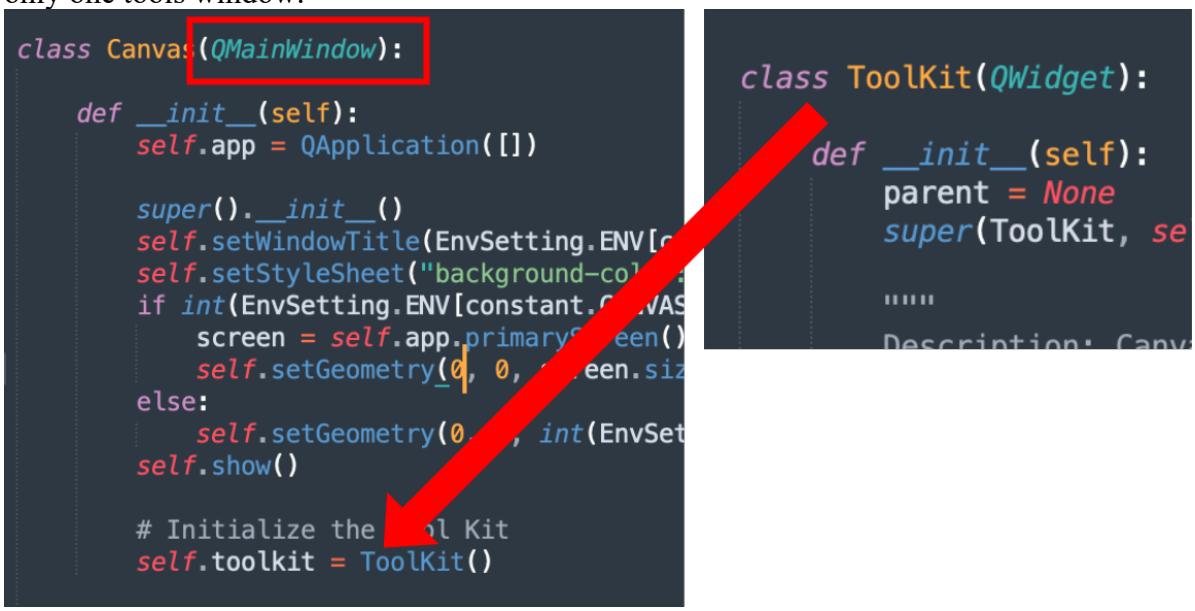
The whole system solely implemented in Python leveraging the PyQt library. Cited online, PyQt is a Python binding for Qt, which is a set of C++ libraries and development tools providing platform-independent abstractions for graphical user interfaces (GUIs), allowing developers to create GUI applications. To put it simply, the PyQt library offer functions for GUI windows, buttons, texts, forms, input fields, painters, brushes, and so on. To ensure compatible with all devices, PyQt6 is chosen as it is still under continuous updates. More information can be found here.

1.1.1 Canvas Class

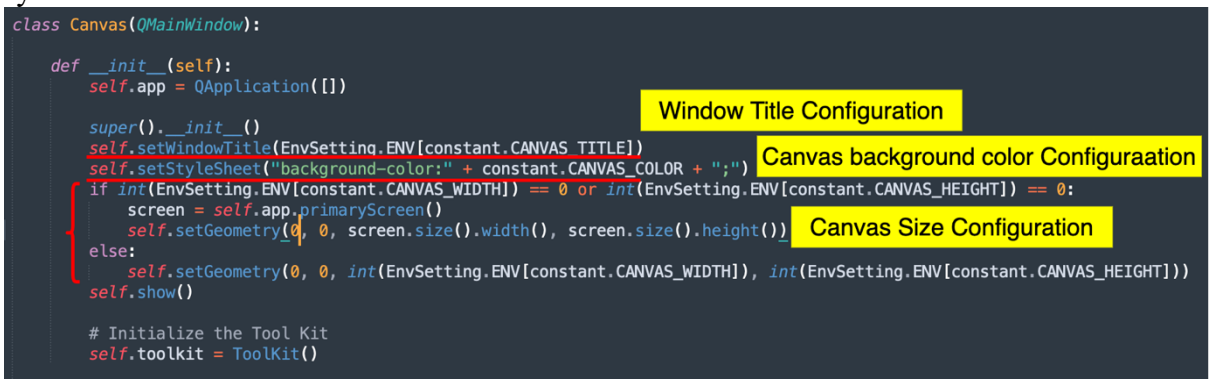
A canvas is created by the main function once executed. In order to create a GUI window, PyQt provides pre-defined *QMainWindow* and *QWidgets* Classes that can be inherit and overwritten. As specified in the official documentation, the *QMainWindow* is a framework for building application UI for main window management.



Since the canvas will be the main window user perform drawings on, it is made to inherit the *QMainWindow* Class. The tools window is then imported and initialized within the canvas Class. The design guarantees that each canvas Class/window will be associated to one and only one tools window.

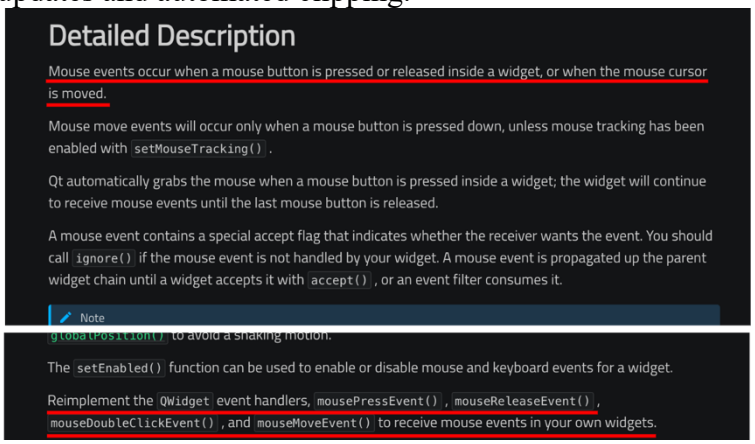


Window configurations are done in the __init__ function. Configurations include window title, canvas color, and canvas size. The settings are stored in either constant or in the env file (refer to section 2 Environment Settings) as parameters. If the size parameters are not set, the system detects the screen size and stretch the window to full screen.



As mentioned, to satisfy intuitive drawing method, the canvas constantly listens on events. This can be accomplished by inheriting and re-implementing the mouse event functions provided by PyQt.QWidgets. In our case, only the *mousePressEvent* and *mouseReleaseEvent* are re-design. Graphics creation is based on combination of the two functions passed to a *paintEvent* function, with repeated updates and automated clipping.

```
def mousePressEvent(self, event): ...

def mouseReleaseEvent(self, event): ...


def paintEvent(self, event): ...
```



1.1.2 ToolKit Class

Similarly, the toolkit Class is also a window with configurations set during initialization, the tools window is further set to always stay above the canvas through a flag. Since the canvas being the main window, the toolkit inherits the *QWidgets* Class only. An additional feature for both canvas and tools window are that the close button is disabled to prevent users from closing window before exiting the system.

```
class ToolKit(QWidget):

    def __init__(self):
        parent = None
        super(ToolKit, self).__init__(parent)

        """
        Description: Canvas is a sub-window created by the main window.
            It is target to stay within the main window
            and Always on top of the main window
        Reference:
            https://stackoverflow.com/questions/70045339/what-is-analog
            https://itechnote
            https://stackoverflow.com/questions/30470433/how-to-put-a-cl
        """

        self.setWindowTitle(EnvSetting.ENV[constant.TOOLKIT_TITLE])
        self.setWindowFlags(QtCore.Qt.WindowType.WindowStaysOnTopHint)

        self.toolkit_width, self.toolkit_height = int(EnvSetting.ENV[cor
```

Flag to stick tools window above all windows

Window Title Configuration

```
        self.show()

    def closeEvent(self, event):
        event.ignore()
```

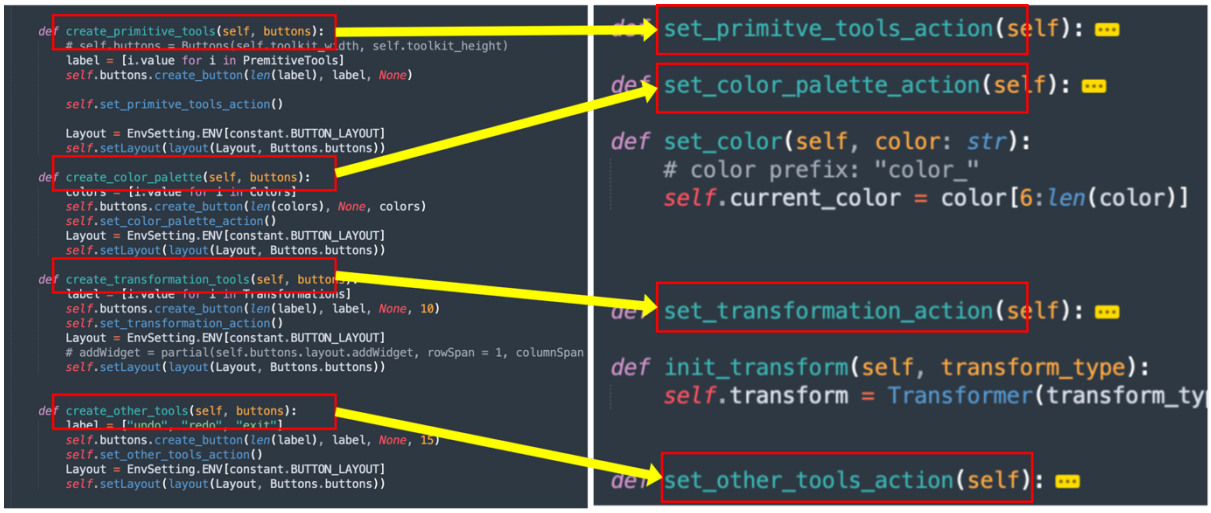
```
        self.toolkit = ToolKit()

    def closeEvent(self, event):
        event.ignore()
```

```
class PremitiveTools(Enum):
    Dot = constant.BUTTON_LABLE_DOT
    Line = constant.BUTTON_LABLE_LINE
    Square = constant.BUTTON_LABLE_SQUARE
    Rectangle = constant.BUTTON_LABLE_RECTANGLE
    Circle = constant.BUTTON_LAYOUT_CIRCLE
    Triangle = constant.BUTTON_LABLE_TRIANGLE

class Colors(Enum):
    Red = "#d00000"
    Blue = "#0000FF"
    Green = "#008000"
    Yellow = "#FFFF00"
    Orange = "#FFA500"
    Black = "#000000"

class Transformations(Enum):
    Translation = "Move"
    Rotation = "Rotate"
    Scaling = "Scale"


class ToolKit(QWidget):
```

According to the functions/features provided, the toolkit calls a Buttons object to create the following buttons (Enum Classes are provide for easy modification if users want to add more buttons):

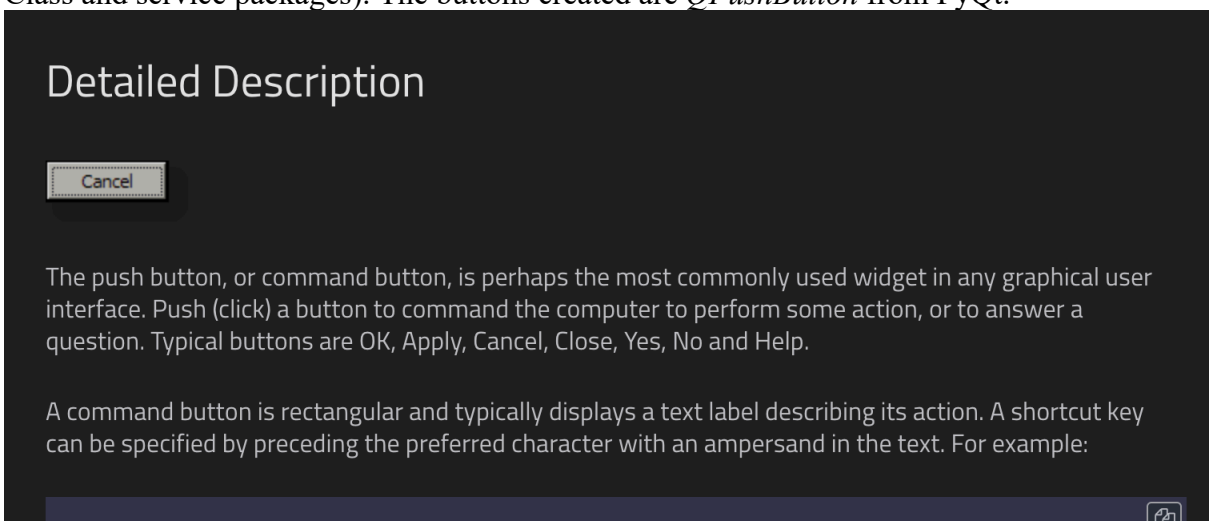
- Primitive tools: buttons for CG primitives
- Color palettes: buttons for coloring
- Transformation tools: buttons for transformation
- Other tools: buttons for additional features

And define the action, operations, and layout of these widgets before rendering on the screen. This is when the action_service and general_service kick in.

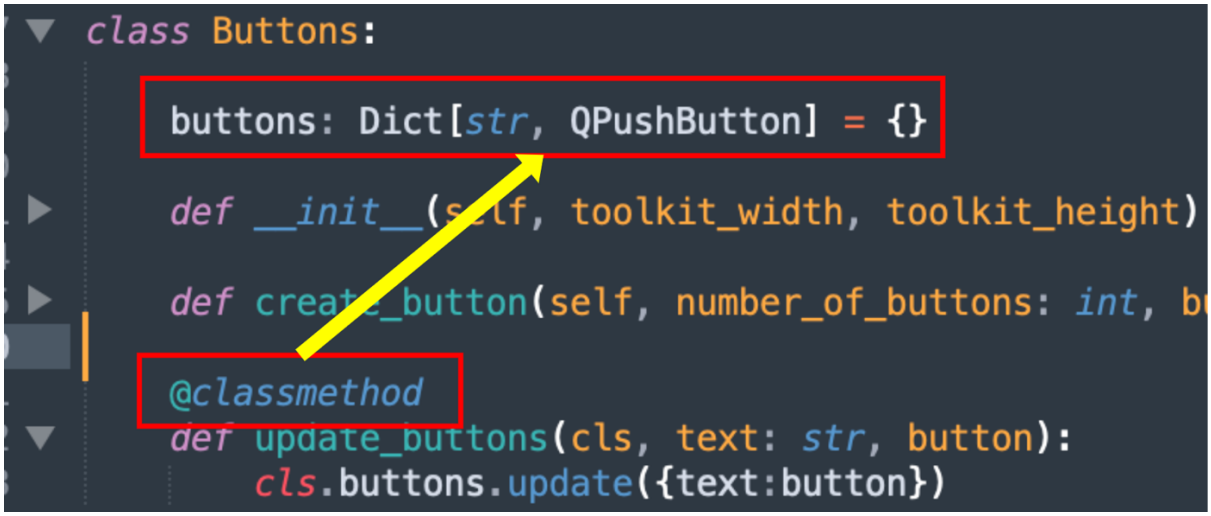


1.1.3 Buttons Class

Compared with the above two classes, the Buttons Class is rather simple. To generalize, the Class is only responsible for creating buttons (Button action and layout are defined in ToolKit Class and service packages). The buttons created are *QPushButton* from PyQt.



Created buttons are stored in a Python dictionary, to prevent initializing Buttons object in different places, and to ensure consistence, a class method is created for updating and accessing buttons. This makes the created buttons associate with the Class instead of the instance.

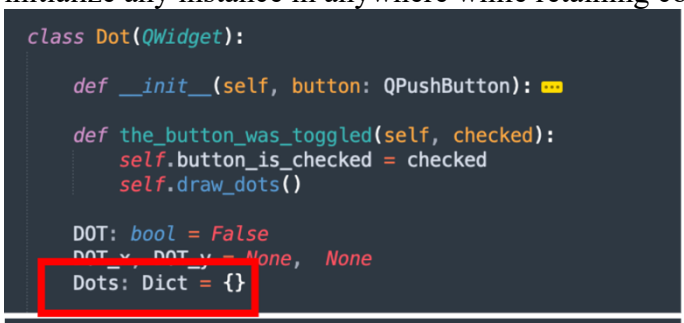


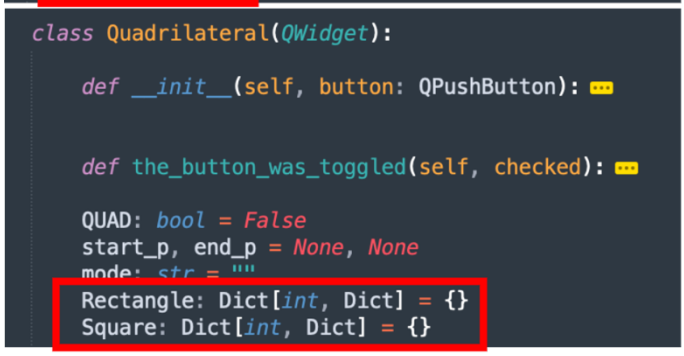
1.1.4 action_service

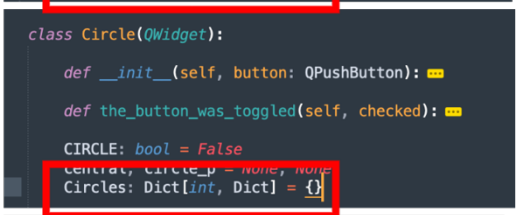
The action_service itself is a package of Classes, including:

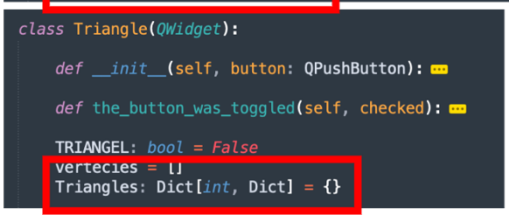
- Dot
- Line
- Quadrilateral
- Circle
- Triangle
- Actions, and
- Transformers

The primitives-related Classes store the graphics and set buttons status by calling functions in general_service. The Classes get event details from Canvas Class and configure the information into corresponding graphics/objects. Applying the same approach, each Class holds a Dictionary (storage) and classmethods (updates) for easy access without the need to initialize any instance in anywhere while retaining consistency.



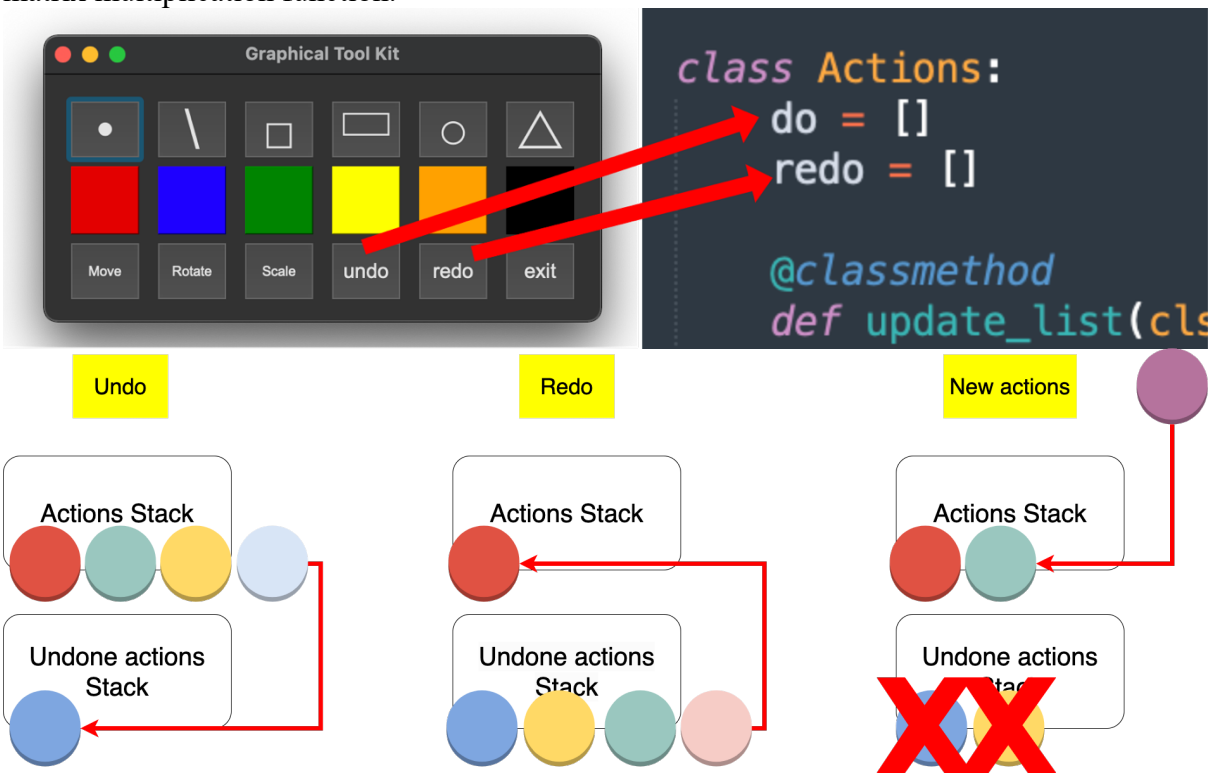






On the other hand, the Actions Class provides services for the undo and redo buttons. Two stacks are implemented to keep track of all user actions. These stacks are also associate with

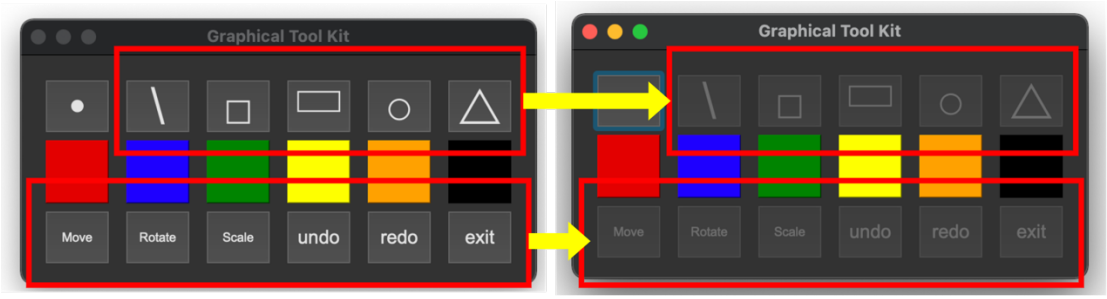
the Class, rather than the instance. In order to simulate the true undo-redo feature, the first stack stores all actions in a last-in-first-out arrangement. When the undo button is clicked, the last action pops out from the first stack and push to the second stack. Clicking on the redo button pops and push actions from second stack to the first. However, whenever a new action is added/performed, this clears the redo stack as a new actions sequence is formed. As for the Transformer, it delivers services related to transformation, such as, configure popup window for user input, matrix formation, retrieving data from primitive-Classes, and trigger the matrix multiplication function.



1.1.5 general_service

The general_service is a package of functions rather than Class(es). Indicated by the name, the package stores functions used in multiple locations. Functions including:

- set_buttons_state:
  The function disables all other buttons when a primitive tool button is clicked. To enable, simply toggle the same button again. This not only avoid conflict of actions but also mimic the actual use case of other CG software.



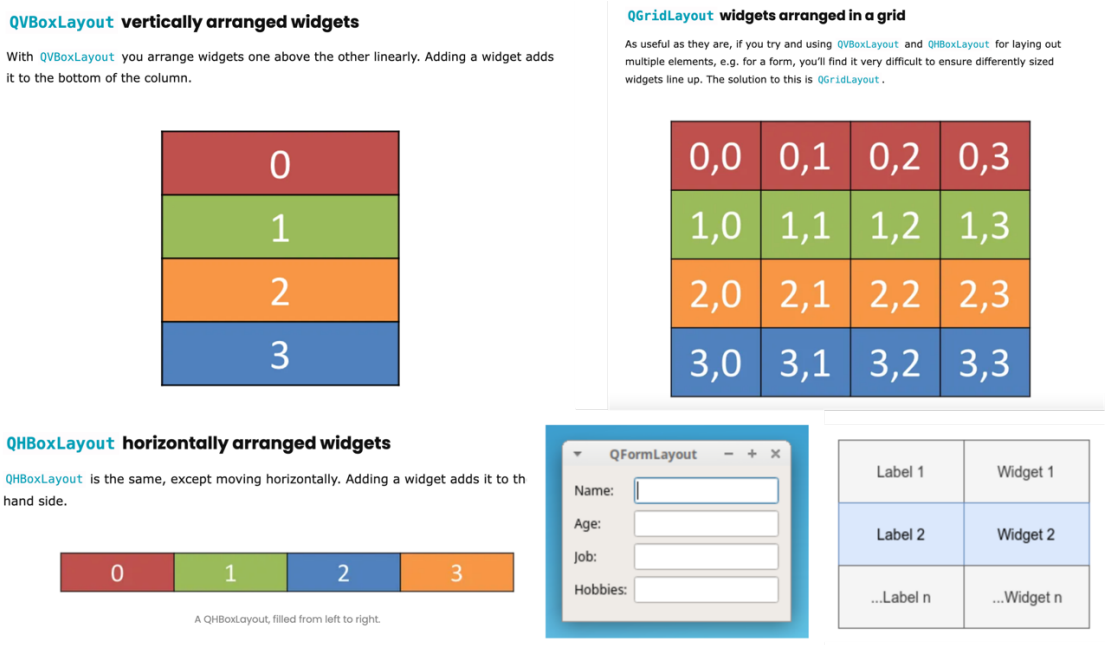
- get_length:
  The function calculates the distance between two given points. It is hardcoded with mathematical equations instead of built in functions.

```
def get_length(start_p: QPoint, end_p: QPoint):
    x_dev = start_p.x() - end_p.x()
    y_dev = start_p.y() - end_p.y()
    return math.sqrt((x_dev ** 2) + (y_dev ** 2))
```

- get_edge:
  get_edge is used to get the width and height of a square or rectangle. Given two points/vertices, the function returns the absolute value of the differences of x coordinates and y coordinates.

```
def get_edge(start_p: QPoint, diagonal_p: QPoint):
    width = abs(start_p.x() - diagonal_p.x())
    height = abs(start_p.y() - diagonal_p.y())
    return width, height
```

- layout:
  The layout here corresponds to the function mentioned in buttons creation. The usage is generalized to set layouts of different widgets of type *QWidgets*. PyQt provides four layouts, namely, *QHBoxLayout*, *QVBoxLayout*, *QGridLayout*, and *QFormLayout*. To summarize, *QHBoxLayout* arrange widgets in a vertical manner, *QVBoxLayout* align widgets against horizontal lines, while *QGridLayout* and *QFormLayout* organize them according to coordinates and pair of rows, respectively.



- matrix_multiplcation
  matrix_multiplcation is to perform transformation on target objects. The matrix will be formed as a result of actions in other Classes before passing in for calculations. Similarly, the calculation is hardcoded instead of utilizing any built-in functions.

```python
def matrix_multiplcation(matrix: List, vertex: List, ref_point: List = [0, 0]):
    #[ matrix[0][0] matrix[0][1] matrix[0][2] ][x]
    #[ matrix[1][0] matrix[1][1] matrix[1][2] ][y]
    #[ matrix[2][0] matrix[2][1] matrix[2][2] ][1]

    v = [\
            matrix[0][0]*(vertex[0] - ref_point[0]) + matrix[0][1]*(vertex[1] - ref_point[1]) + matrix[0][2]*vertex[2] + ref_point[0], \
            matrix[1][0]*(vertex[0] - ref_point[0]) + matrix[1][1]*(vertex[1] - ref_point[1]) + matrix[1][2]*vertex[2] + ref_point[1], \
            matrix[2][0]*(vertex[0] - ref_point[0]) + matrix[2][1]*(vertex[1] - ref_point[1]) + matrix[2][2]*vertex[2] ]

    return QPoint(v[0], v[1])
```

1.1.6 util

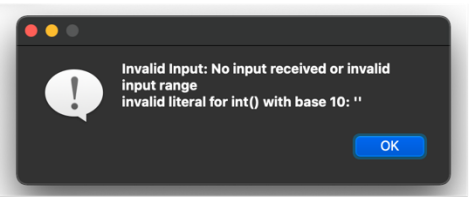
The util package holds small snippets of functions that serve the system as a whole rather than specific functions or classes. Currently, it is used for Environmental Configurations, more detailed will be elaborated in section 2 Environment Settings.

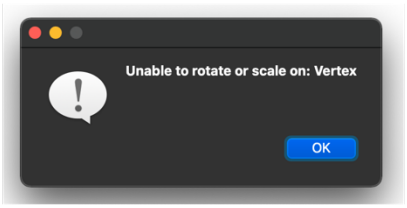
1.1.7 constant

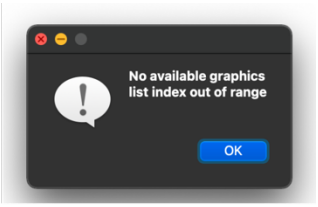
Constants are values or variables that remain unchanged under all condition and throughout the whole execution process. These values can be used globally once import the package.

1.1.8 exception

To prevent unexpected errors shutting down the system, try-except blocks are added here and there to catch exceptions. As a GUI application, especially for CG drawings, the exceptions are design to show error messages in a popup window. Each exception in the exception package inherits the *QWidgets* Class to create a window of type *QMessageBox*. The error messages are captured and embedded in pre-defined texts. The program is freeze-d until the confirmation button is clicked.

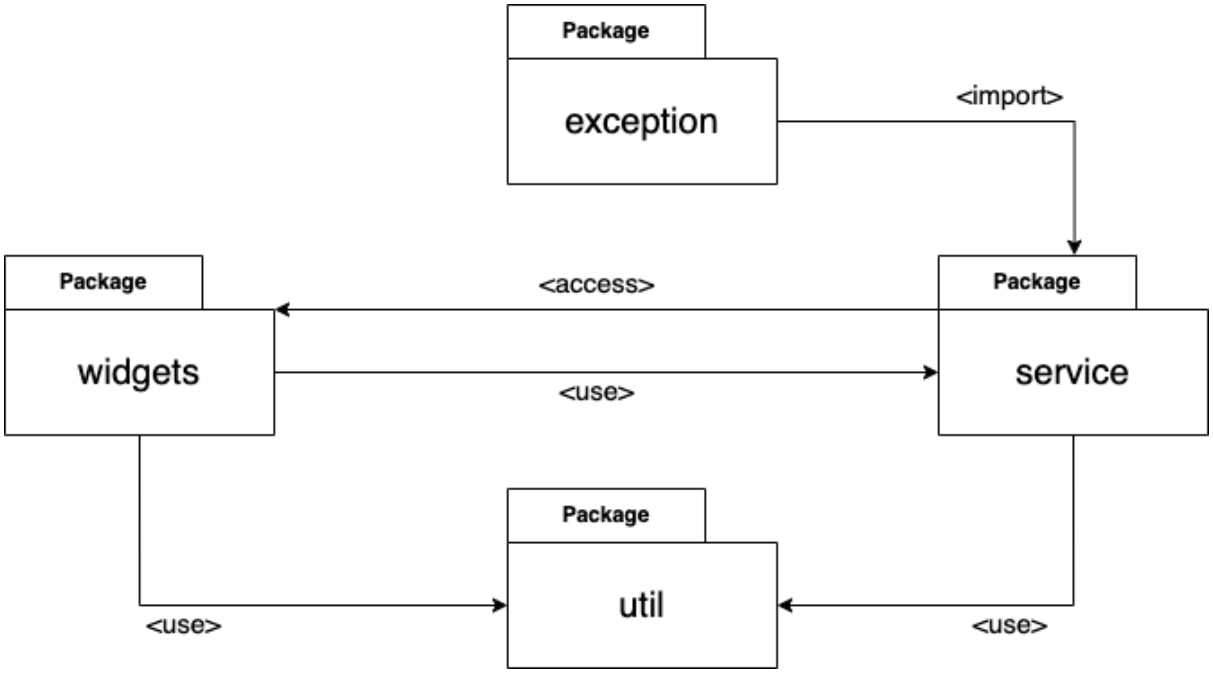




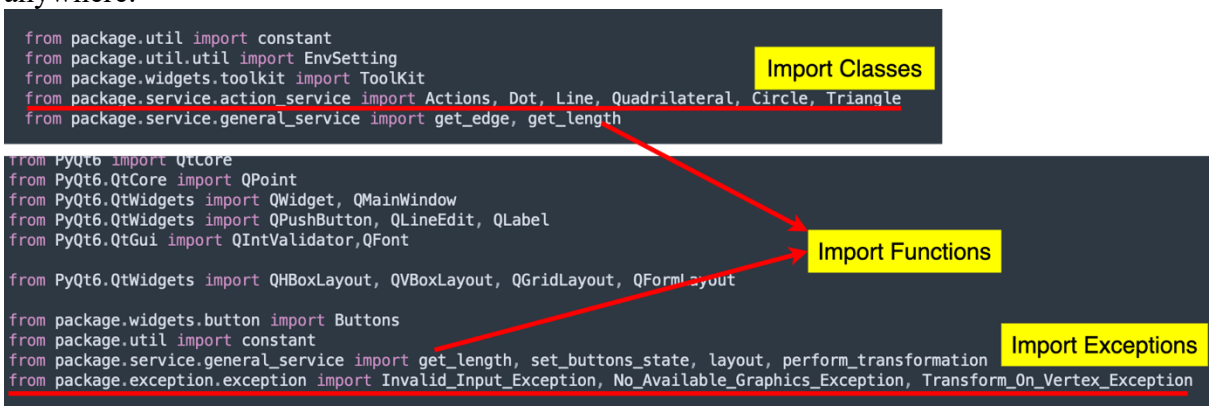


1.2 File Architecture

The File Architecture provides descriptions from a package point of view. Apart from the main.py file, all codes are located under the package folder grouped in different categories. Consider the fact that the Canvas, ToolKit, and Buttons Class are closely related to GUI objects rendering, these files are filed under the widget package. Although windows are also displayed in the exceptions, the main purposes are still to capture errors and notified users of prohibit actions. Hence, it is placed under the exception package. On the other hand, the action_service and general_service are tied together as services while constant and util are both viewed as utils.



Grouping codes in such way won't grant correct access during execution using the import statement. Additional __init__.py files are required in all folders to make all directories a regular package. This is due to the introduction of Namespace packages from Python3.3. The __init__.py will be executed implicitly to bound the scope of objects in the package. Documentations can be found here. Accordingly, Classes and functions can then be imported anywhere.

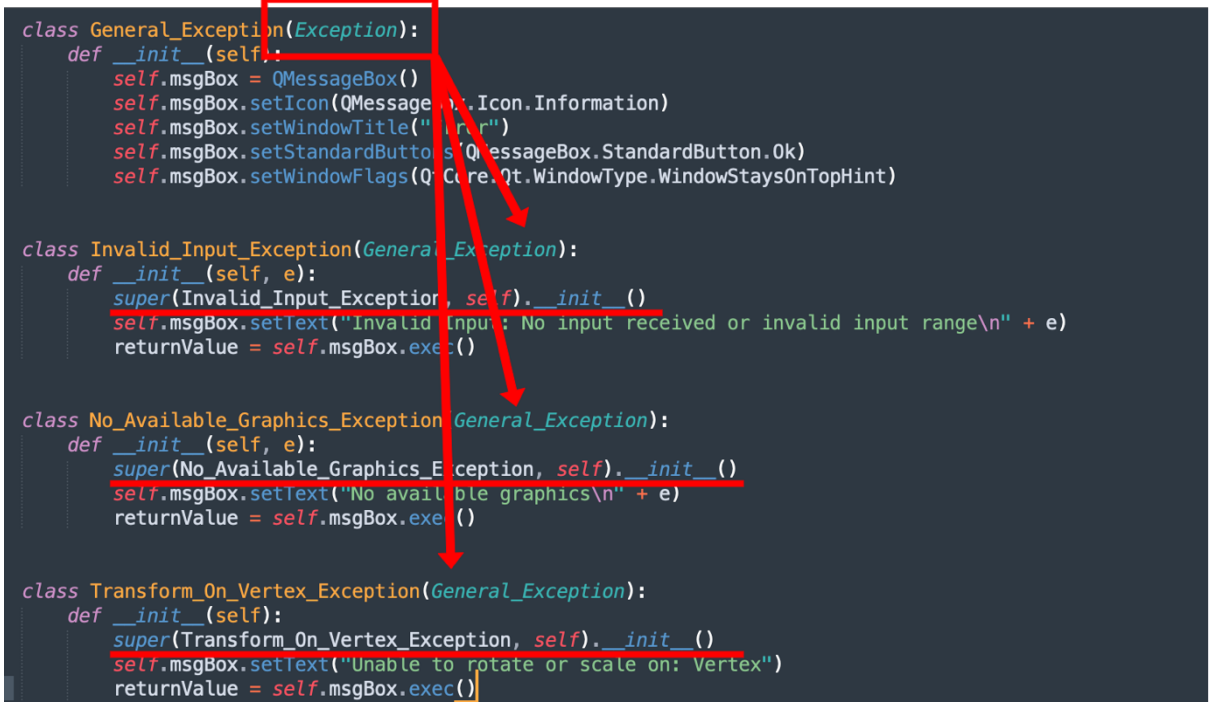


1.3 Program extension and re-write

The system architecture is devised in a way for easy modification and extension in the future. As all components of the code are objects, new features can be implemented easily by adding functions as behaviors of Classes. To be more specific, creating new widgets and operations related to graphics can simply be done by:

- Initializing a Buttons instance and update the buttons dictionary
- Importing pre-define widgets provided by the PyQt library, including message box, forms, input field, scroll bar, etc.
- Utilizing functions in general_service
- Utilizing existing Classes, stacks, or services in action_service

Secondly, as extending the program indicating more errors or exceptions to be handled. The system also granted capabilities to construct new exceptions with minimal effort. All exceptions inherit from a General_Exception, which defines the popup window, icon, titles, flags, and other basic settings. Users simply have to set custom error messages/texts and call the exec() to form new exceptions.



Lastly, for both end users and developers, we extend the control of the system by isolating the window settings and environment configuration from the code into an env file. Not only users can be free from worries of altering the codes unconsciously, but developers can add new parameters and implement them independently.

2 Toolkit Features and Functionalities

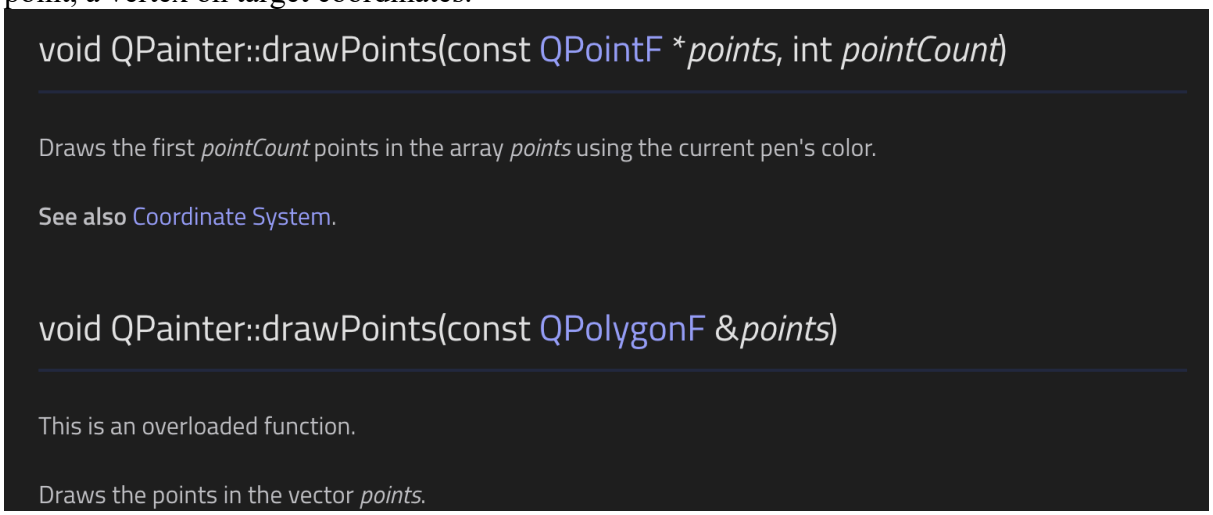
2.1 Graphics and Algorithms

From a product point of view, users are able to:

- Draw a Dot(vertex) on the canvas
- Draw a Line on the canvas
- Draw a Square on the canvas
- Draw a Rectangle on the canvas
- Draw a Circle on the canvas
- Draw a Triangle on the canvas
- Change colors of the objects
- Move, rotate, and scale the objects
- Undo the last action
- Redo the previous action

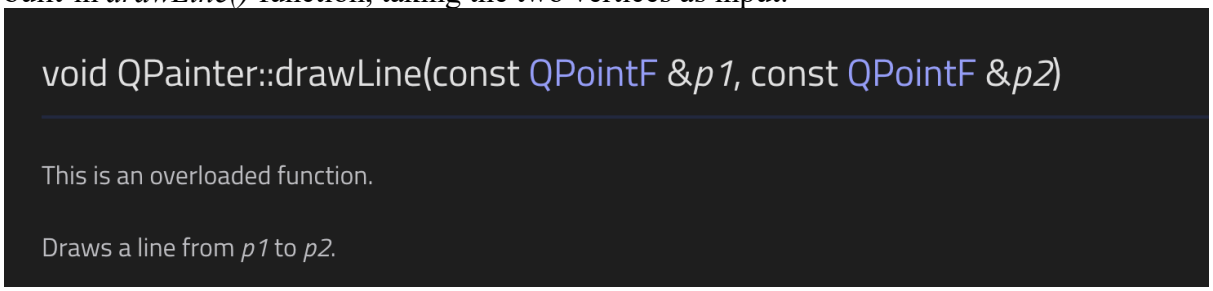
2.1.1 Dot(Vertex) drawing

In order to draw a dot, click on the dot button to trigger the tool. As mentioned, all other buttons will be disabled until user toggles the dot button again. Click on any or desired location on the canvas to draw a dot. The *mousePressEvent* detects the event and gets the cursor coordinates as the vertex's coordinates. In order to switch colors, choose the desired color before toggling the dot button. PyQt provides a *drawPoint()* function which place a point, a vertex on target coordinates.



2.1.2 Line drawing

The Line drawing feature gives user fully control on both the line position and the length of the created line. Choose a starting point and an ending point as the two end points of a line. Click and hold on one of the points, drag to the target position then release your mouse. The *mousePressEvent* and *mouseReleaseEvent* gets the coordinates of your mouse on clicking and releasing as the two vertices. This simulates the intuitive drawing method of a professional CG software, regardless of whether the mouse movement aligns to a straight line, the result depends solely on the two vertices and links them to form a line. This is done by as well a built-in *drawLine()* function, taking the two vertices as input.

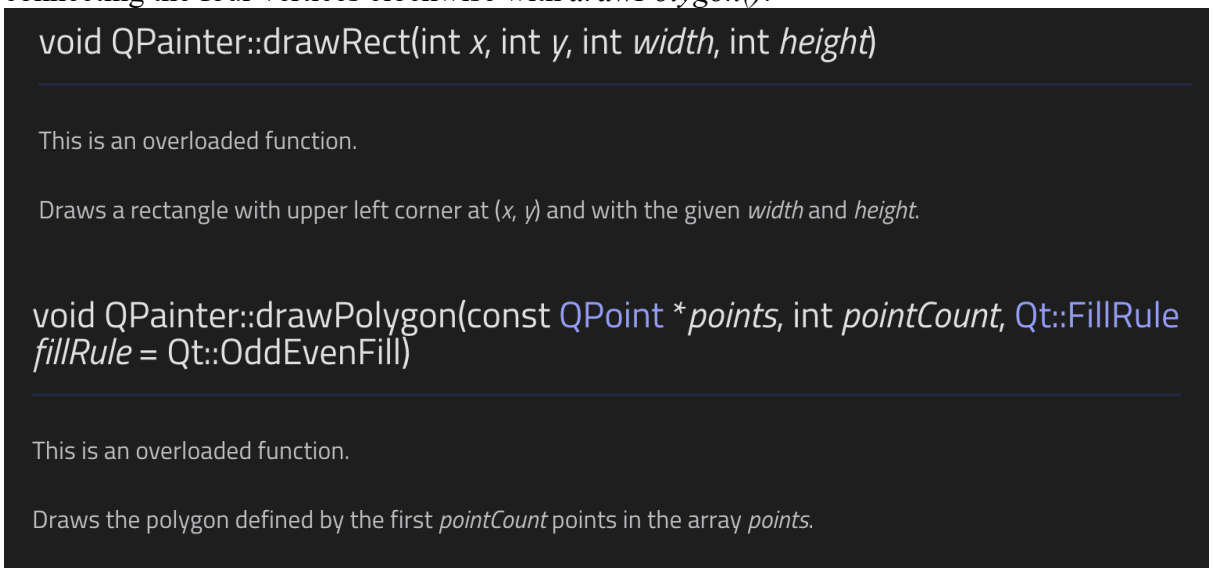


2.1.3 Quadrilateral drawing

The term Quadrilateral refers to only square and rectangles in our context. Drawing squares and rectangles are in similar ways, both by clicking-dragging-and-releasing(dropping). The two coordinates by clicking and releasing represents one of the four vertices and its diagonal vertex. Given the two vertices, the service packages calculate the width and height to generate a quadrilateral perpendicular to the x and y axis. As for drawing squares, the length of edge will be the result of min(width, height).

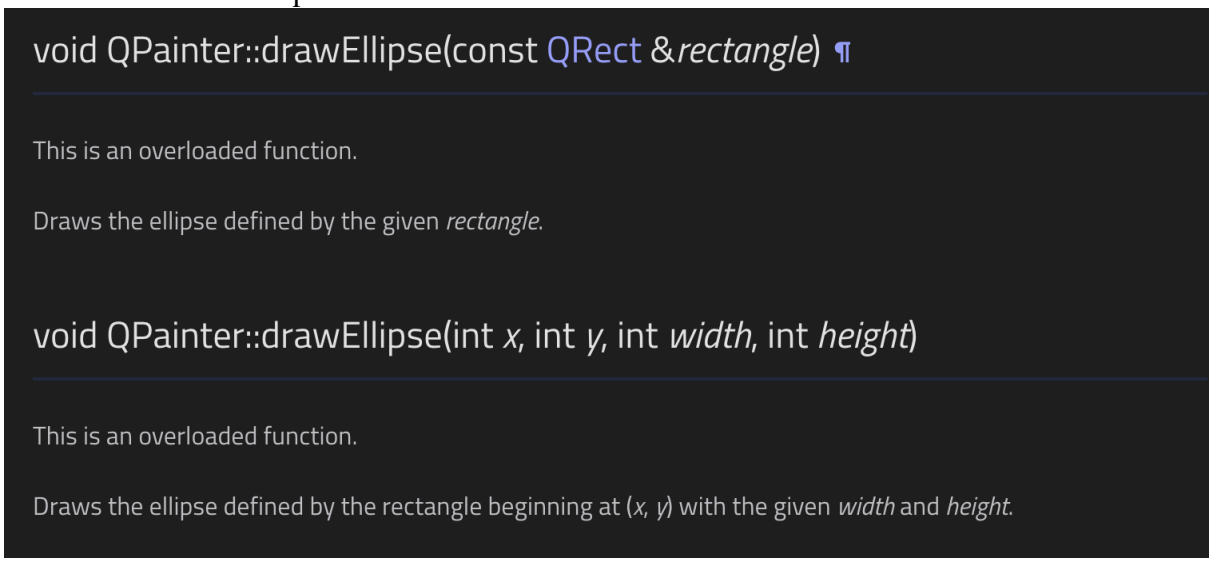
Rendering the results for squares and rectangles is complicated and troublesome. PyQt indeed offer function named *drawRect()* to draw quadrilateral. However, the results are limited to

graphics perpendicular to the two axes, that is, the drawings will be incorrect after rotation. To address the problem, squares and rectangles are instead treated as polygons, drawn by connecting the four vertices clockwise with *drawPolygon()*.



2.1.4 Circle drawing

Circles are also drawn by clicking and dragging. Yet only the clicking coordinate is considered significant since it represents the center/central of a circle. The distance between the central and where the cursor is released is calculated as the radius. With the two components, a circle can be formed. There are no built-in functions to draw a circle in PyQt. Instead, the standard way is to draw an ellipse, using the *drawEllipse()* function, and set the two inner axes to be equal.



2.1.5 Triangle drawing

Theoretically, in a CG context, a triangle is represented by its three coordinates. Specifying the three coordinates allows the computer to render a triangle by connecting the three coordinates in order. Based on the theory, the drawing of a triangle is attained by clicking on three coordinates on the canvas. The *mousePressEvent* handles the three coordinates and passes to the services to form a list. The triangle is painted using the *drawPolygon()* built-in function by PyQt, linking the points in order.

2.1.6 Color changing

The operation for color changing is simple. Since all buttons will be locked during user drawing progress, it is required to select target color before drawing. Currently, there are only 6 colors available, but it can be extended easily in the code.

2.1.7 Transformation

Currently, the program supports only translation, scaling, and rotation, while all of them allows user to specify the transformation factors through input textbox. Translation is performed about the origin. The translation matrix is formed, using a list of lists, in column-wise.

```
if window_title == "Move":
    self.matrix = [\
                [1, 0, int(self.x_inputBox.text())], \
                [0, 1, int(self.y_inputBox.text())], \
                [0, 0, 1]]
```

On the other hand, rotation rotates objects about the starting point of the target object, counterclockwise. The starting point is the mouse clicking point when drawing and generating the object. The rotate angle is received from the user input in degrees. However, the math function sin and cos in Python take radius as input instead of degrees. Thus, before constructing the matrix, the input is mapped to radius via an equation.

```
elif window_title == "Rotate":
    degree = ((360 - int(self.theta_inputBox.text()))*(math.pi/180))
    self.matrix = [\
            [round(math.cos(degree), 1), (-1) * round(math.sin(degree), 1), 0], \
            [round(math.sin(degree), 1), round(math.cos(degree), 1), 0], \
            [0, 0, 1]]
```

As for scaling, it is also done about the starting point. Users are allowed to set different scaling variables to x and y coordinates. The matrix is formed in the following manner.

```
elif window_title == "Scale":
    self.matrix = [\
                [int(self.x_scale.text()), 0, 0], \
                [0, int(self.y_scale.text()), 0], \
                [0, 0, 1]]
```

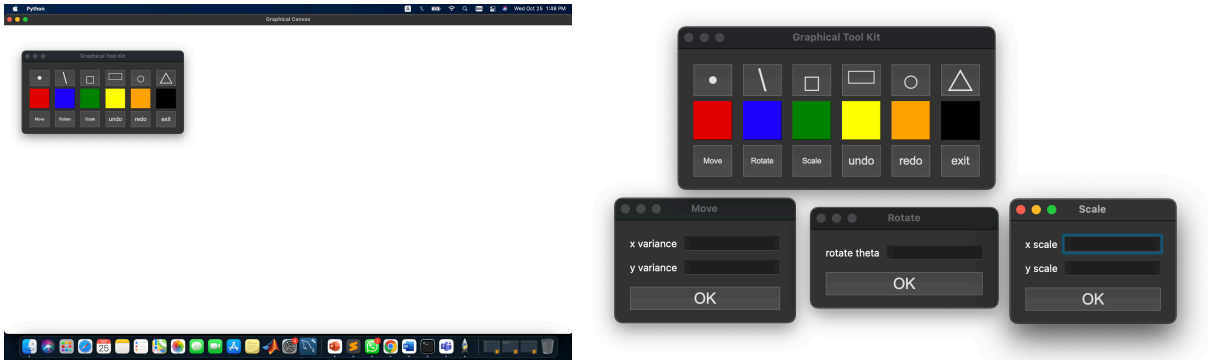
Once the matrix is formed, it is pass along with the reference point to the general_service described in section 1.1.5. If no reference point is available, a default coordinate (0, 0) will be used during matrix multiplication. The target coordinates will be moved to the origin, perform transformation, then move back about the reference coordinate to get the results.

2.2 User Interface

The user interface contains a white, by default, window that allows users to draw with their mouse on, and a relatively small window of buttons. According to the specifications of system design, the tools window sticks and stays on top of all window layers. The idea was to

provide a canvas as big as possible for users to draw on and a load of buttons that can be access easily. Therefore, the canvas was made to stretch according to the physical screen size if not explicitly set, and the buttons float above the canvas window. Even though it covers part of the screen, the issue can be addressed simply by dragging the tools window away.

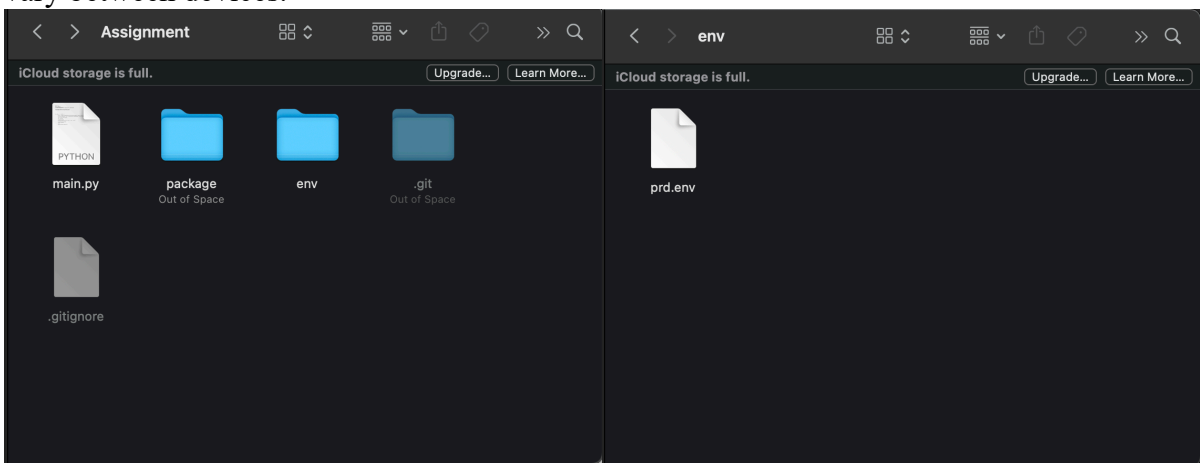
As for the Transformation tools, each button creates a popup window that reads in user inputs. An example of the windows on a Mac device is shown in the following figure.



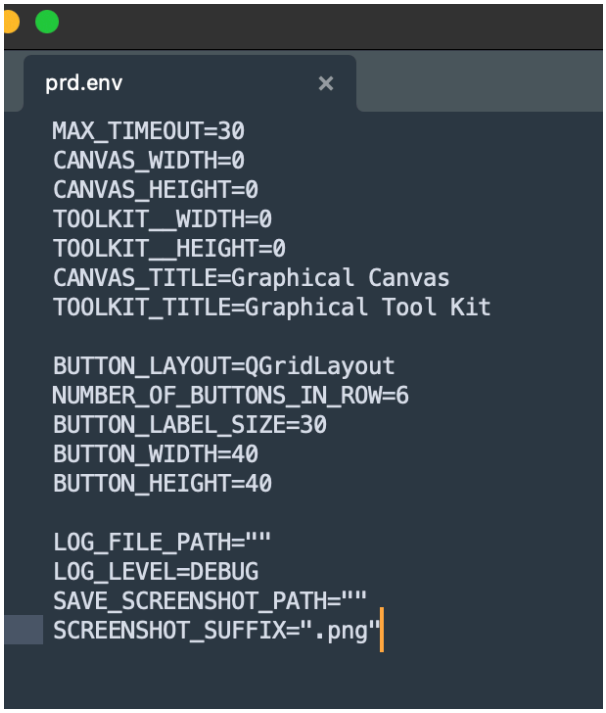
3 Environment Settings

3.1 Environment variables configuration

Environment variables are settings including window size, window titles, customized layout parameters, or other system parameters such as log path size, and screenshot save path. These settings are stored in the env folder located in the root directory of the program. The differences between constant and env parameters are, constants remain unchangeable under all circumstance while env files allow users to modified according to different environments. Secondly, constants are mainly for the program itself rather than the environments that may vary between devices.



Follow the existing settings and amend the value according to users' needs. If new environment variables are required or created, users are responsible for also implementing the logic and updating the constant variables in constant.py class, located in the package/util/ folder. Some possible directions to work on are to implement logging in the system and saving screenshots, the file path can then be configured in the env file.



4 Program execution

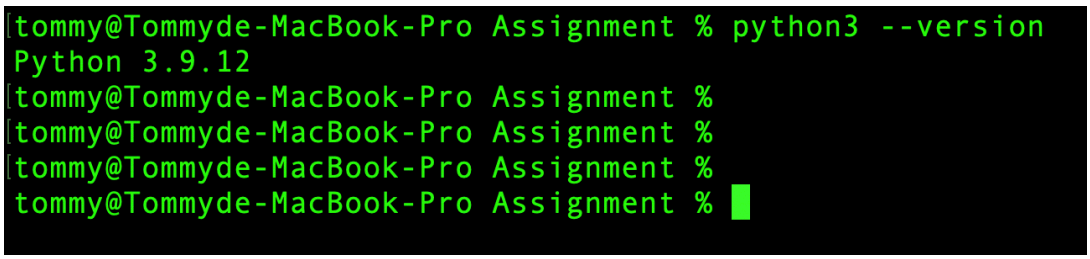
4.1 System setup

4.1.1 Python setup

Download the latest version of python from the official website here for Windows. For Linux and Mac users, visit the links here and here. Follow any installation guides online to install python. Useful links can be found here or here. It is required to install python with minimum version 3.7.6.

Verify the installation by executing the following command in Command Prompt.

```
python3 --version
```



For recent versions, pip is installed along with python during installation. Verify by running the command.

```
pip3 --version
```

```
tommy@Tommyde-MacBook-Pro Assignment % pip3 --version
pip 22.0.4 from /usr/local/lib/python3.9/site-packages/pip (python 3.9)
tommy@Tommyde-MacBook-Pro Assignment %
```

Update the pip to latest version if needed by running the command

```
python -m pip install --upgrade pip
```

If the pip command cannot be recognized, indicating pip is not installed on the machine. Follow the instructions here to install pip via command prompt.

4.1.2 Pyqt6 setup

With pip installed on local pc, run the following command to install the Pyqt6 library

```
pip install pyqt6
```

For Mac users, an alternative is to use the homebrew command, homebrew downloads the latest version of the target package without the need to specify.

```
brew install pyqt
```

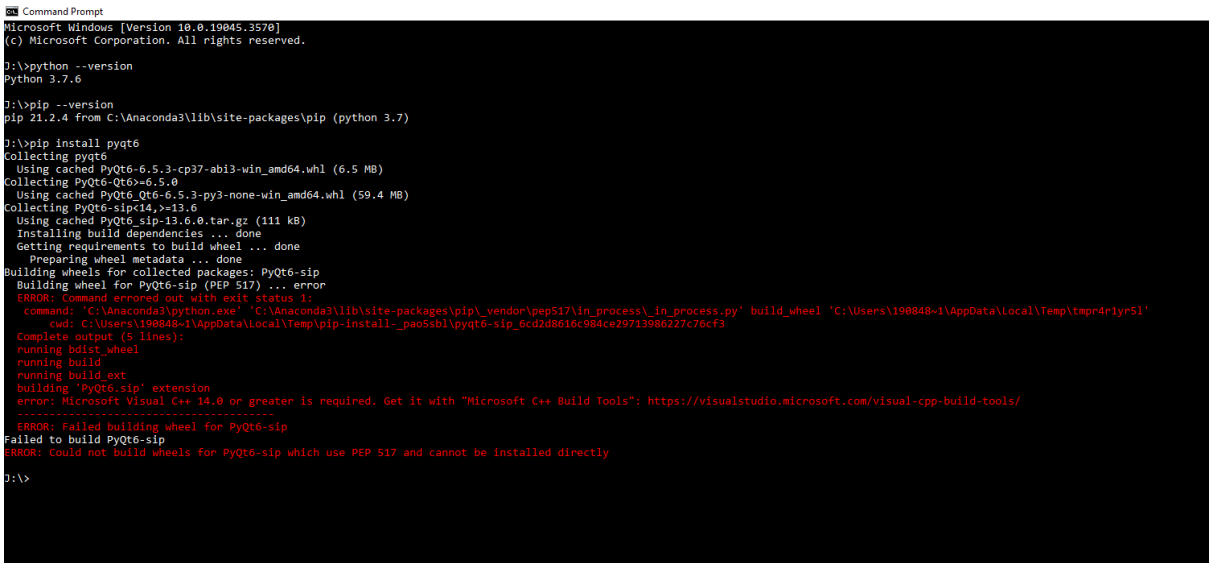
More options can be found in the link here. Finally, verify the installation by the command

```
pip3 show pyqt6
```

A sample output of the above command is as follows.

```
tommy@Tommyde-MacBook-Pro Assignment % pip3 show pyqt6
Name: PyQt6
Version: 6.2.0
Summary: Python bindings for the Qt cross platform application toolkit
Home-page: https://www.riverbankcomputing.com/software/pyqt/
Author: Riverbank Computing Limited
Author-email: info@riverbankcomputing.com
License: GPL v3
Location: /usr/local/lib/python3.9/site-packages
Requires: PyQt6-sip
Required-by: PyQt6-3D, PyQt6-Charts, PyQt6-DataVisualization, PyQt6-NetworkAuth, PyQt6-WebEngine
tommy@Tommyde-MacBook-Pro Assignment %
```

For Window users, an error may occur when installing the PyQt library through command prompt. The error should be something similar to:
error: Microsoft Visual C++ 14.0 or greater is required. Get it with ……



This is due to the fact that the PyQt library depends on C++ and the C++ compiler, and the machine is lacking such dependencies. To solve the problem, the solution is to navigate to the URL specified in the error message and download the visual C++. More information is described in here and here. Once the visual C++ is installed, re-run the command again to install PyQt. An example set up in Window is shown in below.

```
Microsoft Windows [Version 10.0.19045.3570]
(c) Microsoft Corporation. All rights reserved.

J:\>python --version
Python 3.7.6

J:\>pip --version
pip 21.2.4 from C:\Anaconda3\lib\site-packages\pip (python 3.7)

J:\>pip show pyqt6\
WARNING: Package(s) not found: pyqt6\

J:\>pip show pyqt6
Name: PyQt6
Version: 6.5.3
Summary: Python bindings for the Qt cross platform application toolkit
Home-page: https://www.riverbankcomputing.com/software/pyqt/
Author: Riverbank Computing Limited
Author-email: info@riverbankcomputing.com
License: GPL v3
Location: c:\anaconda3\lib\site-packages
Requires: PyQt6-sip, PyQt6-Qt6
Required-by:

J:\>
```

4.2 Execution

Once the setup is ready, open command prompt or terminal and visit the program folder where the file main.py is located.

```
Last login: Mon Oct 23 21:46:44 on ttys000
tommy@Tommyde-MacBook-Pro ~ % cd Desktop/PolyU/COMP4422/Assignment
tommy@Tommyde-MacBook-Pro Assignment % python3 -B main.py
tommy@Tommyde-MacBook-Pro Assignment % ls -l
total 8
drwxr-xr-x@ 3 tommy  staff   96 Oct  5 10:10 env
-rw-r--r--@ 1 tommy  staff  356 Oct 22 11:21 main.py
drwxr-xr-x@ 8 tommy  staff  256 Oct 23 14:44 package
tommy@Tommyde-MacBook-Pro Assignment %
```

The system takes one parameter for running the Toolkit: the env file to be used. As mentioned, users are able to create or customize the environmental variables in a .env file under the env folder. By default, a production env file named prd.env is created in advanced. Users can type in the following command to check for parameter details.

```
python3 -B main.py --help
```

The output should be as follows:

```
tommy@Tommyde-MacBook-Pro Assignment % python3 -B main.py --help
usage: main.py [-h] --env ENV

Computer Graphics ToolKit

optional arguments:
  -h, --help  show this help message and exit
  --env ENV   Mandate, the environment used
```

Parameter name	Description	Param type	Constrain
--env	The environment to be select	string	Compulsory

To start the program/Toolkit, execute the following command

```
python3 -B main.py –env ‹env›
```