

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Інститут прикладного системного аналізу**

«Комп'ютерна графіка»
(назва кредитного модуля)

Звіт до лабораторної роботи 1

Виконав ст. гр ДА81 Переяславський С. К.

Дата 09.09.2020

Керівник Голубова І. А.

Київ – 2020

Зміст

Завдання.....	3
Опис обраної графічної бібліотеки.....	5
UML-діаграма класів.....	6
Короткий опис	7
Блок-схема роботи програми	11
Висновки	12
Лістинг програми.....	13

Завдання

Мета роботи: отримати навички створення графічних програм. Ознайомитись з можливостями OpenGL або обрати іншу графічну бібліотеку.

Завдання:

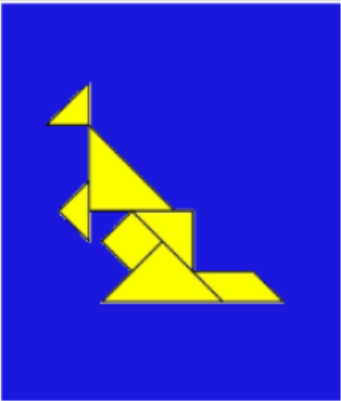
- 1) Ознайомитися з принципами побудови двовимірної системи координат.
- 2) Використовуючи обрану графічну бібліотеку, на основі примітивів зобразити істоту за варіантом з таблиці
- 3) Використовуючи бібліотеку що відповідає за системний рівень операцій вводу-виводу - реалізувати рух істоти у заданому векторі (див. таблицю). Управляючі клавіші - ADWS
- 4) Розібратися з принципами роботи функцій:

```
glViewport,  
glMatrixMode,  
glLoadIdentity,  
glOrtho,  
glClearColor,  
glClear,  
glColor3ub,  
glBegin,  
glEnd,
```

```
glutInit;  
glutInitDisplayMode;  
glutInitWindowSize;  
glutCreateWindow;  
glutDisplayFunc;  
glutReshapeFunc;  
glutKeyboardFunc;  
glutMainLoop;  
glutMouseFunc,  
glutSpecialFunc,  
glutIdleFunc,
```

- 5) Знайти чим представлений аналогічний функціонал(з п.4) у обраній вами графічній бібліотеці

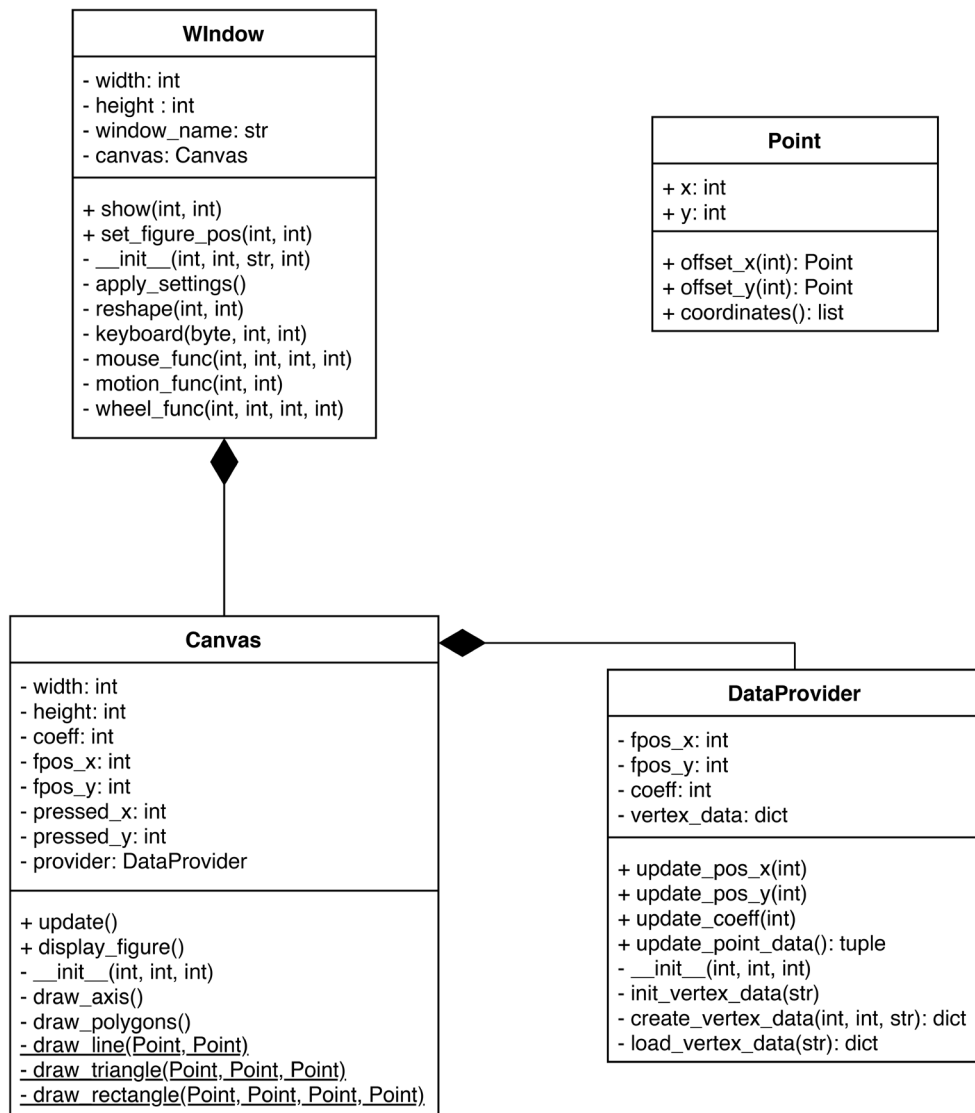
Індивідуальний варіант завдання:

6		$(-1,0)$
---	---	----------

Опис обраної графічної бібліотеки

Для виконання лабораторної роботи була обрана графічна бібліотека OpenGL. Цей вибір був обумовлений тим, що на першому курсі був деякий досвід роботи з OpenGL та ще й було цікаво реалізувати це на мові програмування Python. При налаштуванні бібліотеки я зіткнувся з тим, що, завантажуючи бібліотеку за допомогою менеджера пакетів `pip`, деякі файли були відсутні, тому довелося повністю видалити `PyOpenGL`, знайти такі пакети, які містять відсутні файли та вручну їх встановити.

UML-діаграма класів



Короткий опис

Клас “Window” відповідає за ініціалізацію вікна та обробляє усі події які його стосуються (дії при: зміні розмірів вікна, згортанні/розгортанні вікна, перекритті його іншим вікном, натисканні клавіш WSAD, масштабування за допомогою колеса прокрутки, перетягування фігурки при зажатій клавіші миші).

Клас “Canvas” відповідає за відображення в області вікна. Екземпляр класу “Window” містить екземпляр класу “Canvas”, який не може існувати окремо. Тут визначені методи, які виводять на екран графічні примітиви, тим самим малюючи фігуру. Малювання фігури відбувається у методі draw_figure(). При малюванні задіяна допоміжна структура даних – словник. Він формується шляхом зчитування даних з .json-файлу. Це було зроблено для того, щоб була можливість виводити на екран будь-яку фігуру, дані про яку зберігаються у .json-файлі. Вони виглядають наступним чином:

```
{
    "points": [список координат усіх вершин, з яких складається фігура у вигляді [x, y]],
    "triangles": [список номерів координат вершин з яких складається трикутник у вигляді [вершина_1, вершина_2, вершина_3]],
    "quads": [список номерів координат вершин з яких складається чотирикутник у вигляді [вершина_1, вершина_2, вершина_3, вершина_4]]
}
```

Для малювання фігури згідно мого варіанту використовуються такі дані про неї:

```
{
    "points": [
        [-4.0, 3.0],
        [-3.0, 4.0],
        [-3.0, 3.0],
        [-3.0, 0.0],
        [0.0, 0.0],
        [-4.0, 0.0],
        [-3.0, 1.0],
        [-3.0, -1.0],
        [-1.5, 0.0],
        [-2.5, -1.0],
        [-1.5, -2.0],
```

```

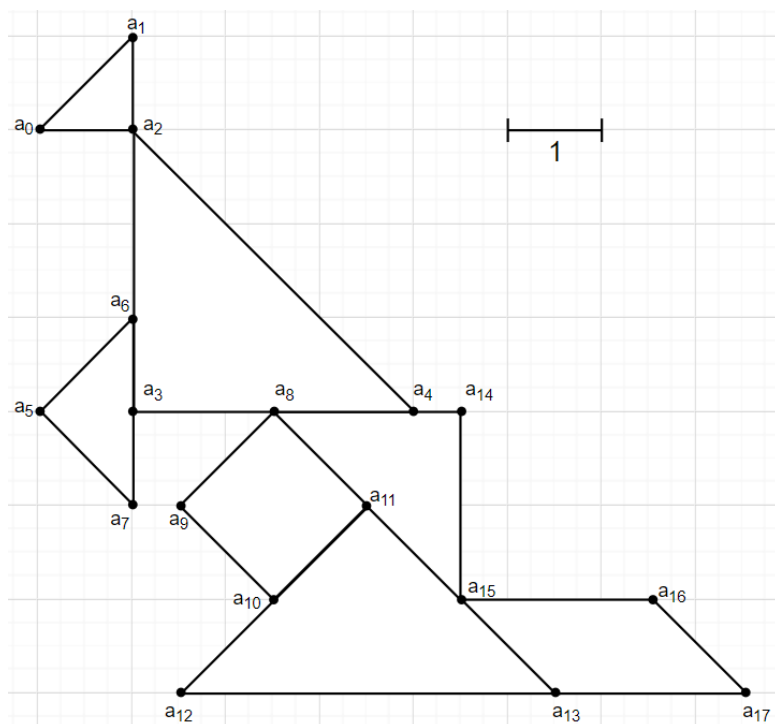
        [-0.5, -1.0],
        [-2.5, -3.0],
        [1.5, -3.0],
        [0.5, 0.0],
        [0.5, -2.0],
        [2.5, -2.0],
        [3.5, -3.0]
    ],
    "triangles": [
        [1, 0, 2],
        [2, 3, 4],
        [6, 5, 7],
        [11, 12, 13],
        [14, 8, 15]
    ],
    "quads": [
        [8, 9, 10, 11],
        [15, 13, 17, 16]
    ]
}

```

Поле “points” містить всі 18 вершин, які використовуються при малюванні фігури, поле “triangles” містить 5 списків номерів вершин, з яких складаються трикутники, поле “quads” містить 2 списки номерів вершин, з яких складаються чотирикутники. Гадаю, що можна сказати що номери вершин виступають посиланнями на точку з координатами у списку “points”. Я обрав такий метод зберігання інформації про фігури, оскільки висунув припущення, що одна й та ж точка може бути використана декілька разів, плюс до того їх може бути дуже багато, тому доцільніше спочатку описати по одному разу всі точки, а потім посилатися на них.

Серед вищевказаних даних власноруч вводилися лише посилання на координати вершин, самі ж координати були обчислені за допомогою структури даних “Point”, а саме: завдяки реалізованим методам та перевизначених операторів додавання та віднімання. На аркуші зошиту була намальована фігура, кожній вершині була співставлена назва.

Фігура на аркуші паперу:



Логіка використання структури даних “Point” така:

- задаються координати початкової точки;
- на основі зміщень від даної точки, навколо неї розташовуються інші точки;

Розглянемо це на прикладі фігури індивідуального варіанту.

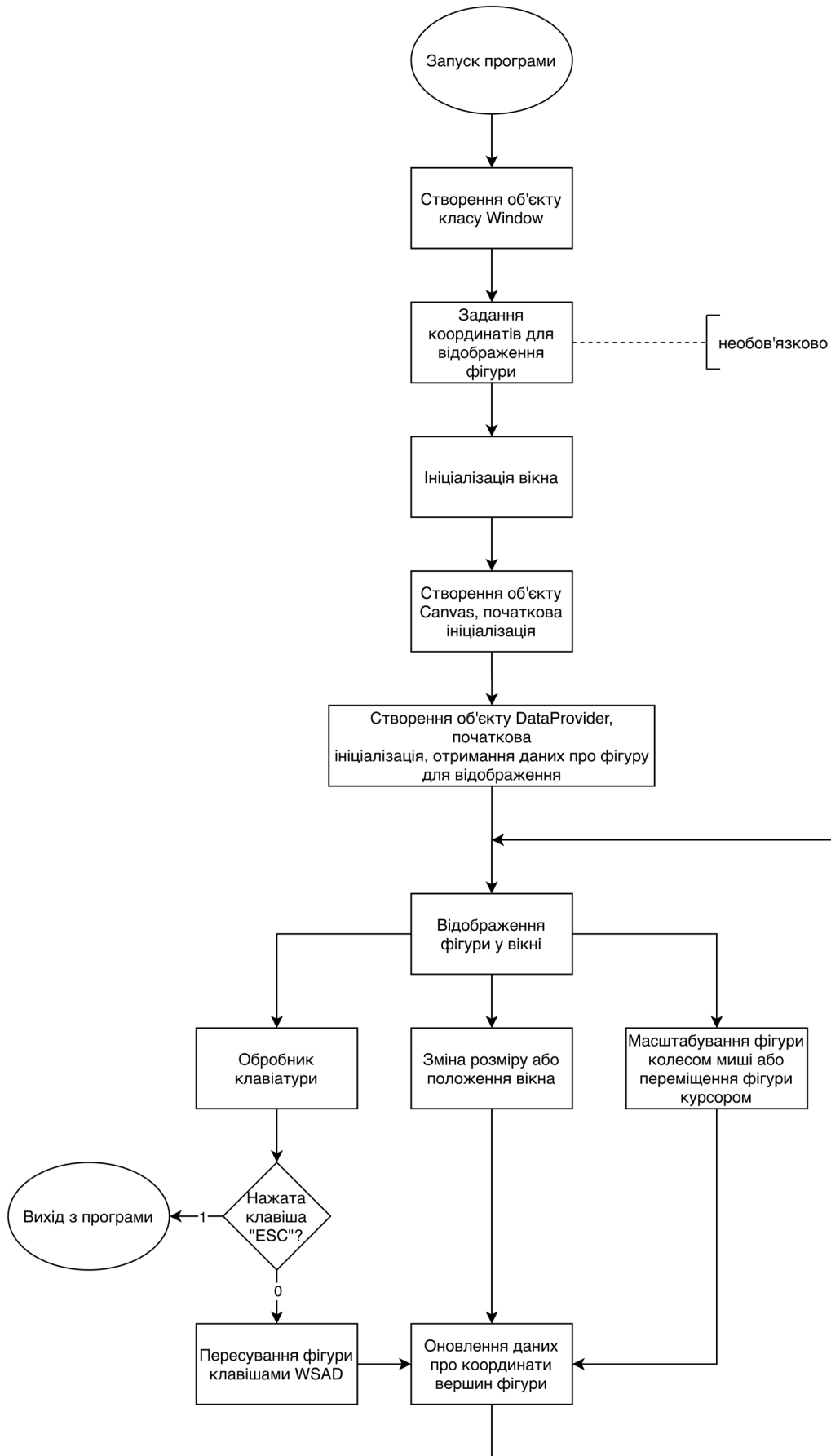
Початкова точка - a_0 , вона єдина точка, для якої суворо задаються координати x та y . Від неї на один одиничний відрізок вправо та на один одиничний відрізок вгору (тобто по діагоналі на 45° в сторону північного сходу відкладається відрізок довжиною: $\sqrt{2} * (\text{одиничний відрізок})$). Відкладена точка є точкою a_1 . Точка a_2 розташовується на одиничний відрізок вправо від точки a_0 або на одиничний відрізок вниз від точки a_1 (вибрати одне з двох). Точка a_3 розташовується на три одиничних відрізків нижче, ніж точка a_2 і т. д. поки не відкладемо останню точку a_{17} .

Всі ці точки є екземплярами класу “Point” і вони мають поля x та y . Ці поля будуть використані при відображенні графічних примітивів.

Клас “DataProvider” відповідає за отримання/створення даних про фігуру, оновлення інформації про координати вершин (при масштабуванні фігури, при

зміні її положення та при будь-яких діях, пов'язаних з вікном та областю малювання).

Блок-схема роботи програми



Висновки

У ході виконання лабораторної роботи була розроблена програма, яка може відображати 2D-фігури. Для її роботи була обрана бібліотека OpenGL. Для реалізації були використані функції з бібліотеки PyOpenGL.GLUT та PyOpenGL.GL. Були описані класи для розподілення зв'язності коду.

Лістинг програми

```
# importing all the needed libraries
from OpenGL.GL import *
from OpenGL.GLUT import *
import sys
import json
# constant global variables
# if INPUT_FILE string is empty (') the information
about figure will be created
# else get data from the given file
INPUT_FILE = 'vertex_data_upd.json'
WRITE_TO_FILE = False
# if MOVEMENT_MODE is 0, a figure can move in all
direction
# else figure can move only along vector (-1, 0)
MOVEMENT_MODE = 0

class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        if type(other) is int or type(other) is float:
            return Point(self.x + other, self.y + other
        )
        else:
            return Point(self.x + other.x, self.y +
other.y)

    def __sub__(self, other):
        return self.__add__(-other)
        # the logic above is equal to logic below
        # if type(other) is int or type(other) is float
        :
        #     return Point(self.x - other, self.y -
other)
        # else:
        #     return Point(self.x - other.x, self.y -
other.y)

    def __str__(self):
        # overriding a string method
```

```
        # used at the beginning of the development
        return '(' + str(self.x) + ', ' + str(self.y)
    ) + ')'
```

```
def offset_x(self, offset_x):
    return Point(self.x + offset_x, self.y)
```

```
def offset_y(self, offset_y):
    return Point(self.x, self.y + offset_y)
```

```
def coordinates(self):
    return [self.x, self.y]
```

```
class DataProvider:
```

```
def __init__(self, fpos_x, fpos_y, coeff):
    self.fpos_x = fpos_x
    self.fpos_y = fpos_y
    self.coeff = coeff
    self.vertex_data = None
    self.init_vertex_data()
```

```
def update_pos_x(self, new_x):
    self.fpos_x = new_x
```

```
def update_pos_y(self, new_y):
    self.fpos_y = new_y
```

```
def update_coeff(self, new_coeff):
    self.coeff = new_coeff
```

```
def update_point_data(self):
    # return new information about the points after
    # changing of any among x coord, y coord, scale
    coefficient
    points = tuple(
        (self.coeff * px + self.fpos_x, self.coeff
* py + self.fpos_y)
        for px, py in self.vertex_data['points']
    )
    return tuple(
        Point(point[0], point[1])
        for point in points
    )
```

```

def init_vertex_data(self, input_file=INPUT_FILE):
    # getting the information about the figure
    if input_file:
        self.vertex_data = self.load_vertex_data(
input_file)
    else:
        # remembering an old coefficient
        temp_coeff = self.coeff
        # setting coefficient to 1 (the default
width of segment)
        self.update_coeff(1)
        # getting a data about vertices coordinates
        #-4 and 3 are the parameters that determine
the centre of figure
        self.vertex_data = self.create_vertex_data
(-4, 3)

        # setting the remembered coefficient back
        self.update_coeff(temp_coeff)

def create_vertex_data(self, x, y, write_to_file=
WRITE_TO_FILE):
    # create data using the offsets from the start
point (a0)
    a0 = Point(x, y)
    a1 = a0 + self.coeff
    a2 = a0.offset_x(self.coeff)
    a3 = a2.offset_y(-3 * self.coeff)
    a4 = a3.offset_x(3 * self.coeff)
    a5 = a3.offset_x(-self.coeff)
    a6 = a3.offset_y(self.coeff)
    a7 = a3.offset_y(-self.coeff)
    a8 = a4.offset_x(-1.5 * self.coeff)
    a9 = a8 - self.coeff
    a10 = a8.offset_y(-2 * self.coeff)
    a11 = a8.offset_x(self.coeff).offset_y(-self.
coeff)
    a12 = a10 - self.coeff
    a13 = a12.offset_x(4 * self.coeff)
    a14 = a8.offset_x(2 * self.coeff)
    a15 = a14.offset_y(-2 * self.coeff)
    a16 = a15.offset_x(2 * self.coeff)
    a17 = a13.offset_x(2 * self.coeff)
    vertices = (a0, a1, a2, a3, a4, a5, a6, a7, a8
, a9, a10, a11, a12, a13, a14, a15, a16, a17)

```

```

# place data into dictionary
vertex_data = {
    'points': [],
    "triangles":
    [
        [1, 0, 2],
        [2, 3, 4],
        [6, 5, 7],
        [11, 12, 13],
        [14, 8, 15]
    ],
    "quads":
    [
        [8, 9, 10, 11],
        [15, 13, 17, 16]
    ]
}
# write points coordinates to a list in a
dictionary
for vertex in vertices:
    vertex_data['points'].append(vertex.
coordinates())
# write information to .json file if necessary
if write_to_file:
    with open('vertex_data.json', 'w') as fout:
        json.dump(vertex_data, fout)
# return dictionary with information about the
figure
return vertex_data

def load_vertex_data(self, file_in):
    # load information about the figure from the .
    json file if necessary
    with open(file_in, 'r') as fin:
        vertex_data = json.load(fin)
    # return dictionary with information about
    the figure
    return vertex_data

class Canvas:

    def __init__(self, width, height, coeff, grid_coeff
):
        self.width = width

```



```

        self.height = height
        self.coeff = coeff
        self.grid_coeff = grid_coeff
        self.fpos_x = width / 2
        self.fpos_y = height / 2
        self.pressed_x = 0
        self.pressed_y = 0
        self.provider = DataProvider(self.fpos_x, self.
fpos_y, self.coeff)

```

```

    def draw_axis(self):
        # drawing grid lines with given step value
        step = self.grid_coeff
        # drawing central axis
        glLineWidth(2)
        Canvas.draw_line(Point(self.width/2, 0), Point(
self.width/2, self.height), True, False)
        Canvas.draw_line(Point(0, self.height/2), Point
(self.width, self.height/2), False, True)
        glLineWidth(1)
        # drawing grid in all directions
        glBegin(GL_LINES)
        i = self.width/2
        while i < self.width:
            Canvas.draw_line(Point(i, 0), Point(i, self
.height), False, False)
            i += step
        i = self.width/2
        while i > 0:
            Canvas.draw_line(Point(i, 0), Point(i, self
.height), False, False)
            i -= step
        i = self.height/2
        while i < self.height:
            Canvas.draw_line(Point(0, i), Point(self.
width, i), False, False)
            i += step
        i = self.height/2
        while i > 0:
            Canvas.draw_line(Point(0, i), Point(self.
width, i), False, False)
            i -= step
        glEnd()

```

```

    def draw_polygons(self):

```

```

        # vertex is a list of points (Point(x,y))
        vertex = self.provider.update_point_data()

        # triangle is a list of vertices numbers that
it consists of
        # len(triangle) = 3
        for triangle in self.provider.vertex_data['
triangles']:
            Canvas.draw_triangle(
                vertex[triangle[0]],
                vertex[triangle[1]],
                vertex[triangle[2]]
            )

        # quad is a list of vertices numbers that it
consists of
        # len(quad) = 4
        for quad in self.provider.vertex_data['quads']:
            Canvas.draw_rectangle(
                vertex[quad[0]],
                vertex[quad[1]],
                vertex[quad[2]],
                vertex[quad[3]]
            )

def display_figure(self):
    glClear(GL_COLOR_BUFFER_BIT)

    # draw grid
    glPolygonMode(GL_FRONT, GL_LINE)
    glColor3f(0.2, 0.9, 0.7)
    self.draw_axis()

    # draw filled polygons
    glPolygonMode(GL_FRONT, GL_FILL)
    glColor3f(0.4, 1.0, 0.6)
    self.draw_polygons()

    # draw only contours of polygons
    glPolygonMode(GL_FRONT, GL_LINE)
    glColor3f(0.0, 0.0, 0.0)
    self.draw_polygons()

    glFinish()

```

```

    def update(self):
        # update provider filed in order to update
        point data
        self.provider.update_pos_x(self.fpos_x)
        self.provider.update_pos_y(self.fpos_y)
        self.provider.update_coeff(self.coeff)

    @staticmethod
    def draw_line(point1, point2, begin=True, end=True
):
        if begin:
            glBegin(GL_LINES)
            glVertex2f(point1.x, point1.y)
            glVertex2f(point2.x, point2.y)
        if end:
            glEnd()

    @staticmethod
    def draw_triangle(point1, point2, point3, begin=
True, end=True):
        if begin:
            glBegin(GL_TRIANGLES)
            glVertex2f(point1.x, point1.y)
            glVertex2f(point2.x, point2.y)
            glVertex2f(point3.x, point3.y)
        if end:
            glEnd()

    @staticmethod
    def draw_rectangle(point1, point2, point3, point4,
begin=True, end=True):
        if begin:
            glBegin(GL_QUADS)
            glVertex2f(point1.x, point1.y)
            glVertex2f(point2.x, point2.y)
            glVertex2f(point3.x, point3.y)
            glVertex2f(point4.x, point4.y)
        if end:
            glEnd()

class Window:

    def __init__(self, width, height, window_name,
coeff=10, grid_coeff=30):

```

```

        self.width = width
        self.height = height
        self.window_name = window_name
        self.canvas = Canvas(width, height, coeff,
grid_coeff)

    def apply_settings(self):
        # bg color, viewport settings and (0, 0)
position
        glClearColor(0, 0.5, 0.5, 1)
        glViewport(0, 0, self.width, self.height)
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        glOrtho(0, self.width, 0, self.height, 1.0, -1.
0)

    def show(self, pos_x=0, pos_y=0):
        glutInit(sys.argv)
        glutInitDisplayMode(GLUT_RGBA)
        glutInitWindowSize(self.width, self.height)
        glutInitWindowPosition(pos_x, pos_y)
        glutCreateWindow(self.window_name)
        # bg color, viewport settings and (0, 0)
position
        self.apply_settings()
        # set callback functions
        glutDisplayFunc(self.canvas.display_figure)
        glutReshapeFunc(self.reshape)
        glutKeyboardFunc(self.keyboard)
        glutMouseFunc(self.mouse_func)
        glutMotionFunc(self.motion_func)
        glutMouseWheelFunc(self.wheel_func)
        glutMainLoop()

    def set_figure_pos(self, x, y):
        # placing a figure in the beginning of
execution
        self.canvas.fpos_x = x
        self.canvas.fpos_y = y

    def reshape(self, w, h):
        # compute a new figure position in reshaped
window
        self.canvas.fpos_x = w * self.canvas.fpos_x /
self.canvas.width

```

```

        self.canvas.fpos_y = h * self.canvas.fpos_y /
self.canvas.height
        # new width and height of reshaped window
        self.width = self.canvas.width = w
        self.height = self.canvas.height = h
        # viewport settings and (0, 0) position
        self.apply_settings()
        # apply changes
        self.canvas.update()

    def keyboard(self, key, x, y):
        pressed_key = key.decode("utf-8").lower()
        if pressed_key == chr(27):
            sys.exit(0)
        if pressed_key == ' ':
            self.canvas.fpos_x = self.width / 2
            self.canvas.fpos_y = self.height / 2
        if MOVEMENT_MODE:
            if pressed_key == 'w' or pressed_key == 's'
:
                if 0 < self.canvas.fpos_x:
                    self.canvas.fpos_x -= self.canvas.
grid_coeff
            elif pressed_key == 'a' or pressed_key == '
d':
                if self.canvas.fpos_x < self.width:
                    self.canvas.fpos_x += self.canvas.
grid_coeff
            else:
                if pressed_key == 'w':
                    if self.canvas.fpos_y + self.canvas.
grid_coeff < self.height:
                        self.canvas.fpos_y += self.canvas.
grid_coeff
                elif pressed_key == 's':
                    if self.canvas.fpos_y - self.canvas.
grid_coeff > 0:
                        self.canvas.fpos_y -= self.canvas.
grid_coeff
                elif pressed_key == 'd':
                    if self.canvas.fpos_x + self.canvas.
grid_coeff < self.width:
                        self.canvas.fpos_x += self.canvas.
grid_coeff
                elif pressed_key == 'a':

```

```

        if self.canvas.fpos_x - self.canvas.
grid_coeff > 0:
            self.canvas.fpos_x -= self.canvas.
grid_coeff
        self.canvas.update()
        glutPostRedisplay()

    def mouse_func(self, button, state, x, y):
        # remember coordinates when mouse button was
pressed
        self.canvas.pressed_x = x
        self.canvas.pressed_y = y

    def motion_func(self, x, y):
        # compute the distance the pointer has gone
        dx = x - self.canvas.pressed_x
        dy = y - self.canvas.pressed_y
        # move figure coordinates
        if 0 < self.canvas.fpos_x + dx < self.canvas.
width:
            self.canvas.fpos_x += dx
        if 0 < self.canvas.fpos_y - dy < self.canvas.
height:
            self.canvas.fpos_y -= dy
        # remember coordinates when pointer is moving
        self.canvas.pressed_x = x
        self.canvas.pressed_y = y
        # apply changes
        self.canvas.update()
        # call display function
        glutPostRedisplay()

    def wheel_func(self, wheel, direction, x, y):
        # if mouse wheel is moving forward zoom in
        if direction > 0 and self.canvas.coeff < 60:
            self.canvas.coeff += 5
        # if mouse wheel is moving backward zoom out
        elif direction < 0 and self.canvas.coeff > 5:
            self.canvas.coeff -= 5
        # apply changes
        self.canvas.update()
        # call display function
        glutPostRedisplay()

```

```
if __name__ == '__main__':  
    window1 = Window(512, 512, 'Lab1 Sergey  
Pereyaslavskiy', coeff=30, grid_coeff=30)  
    window1.show()
```