TP4 This exercise aims to focus on handling mouse events and drawing on images using OpenCV functions. You will find theoretical explanation and code to complete.

1. Handling Mouse Events

OpenCV supports for detecting mouse events. Mouse events include mouse clicks and movements over an attached OpenCV window.

To detect mouse events is necessary to define a callback function in the OpenCV C++ code attaching to the OpenCV window. That callback function will be called every time, mouse events occur. That callback function will also give the coordinates of the mouse events. (e.g - (x, y) coordinate of a mouse click).

This means that, to enable response to mouse clicks, we must first write a callback routine that OpenCV can call whenever a mouse event occurs. Once we have done that, we must register the callback with OpenCV, thereby informing OpenCV that this is the correct function to use whenever the user does something with the mouse over a particular window.

The callback can be any function that takes the correct set of arguments and returns the correct type. Here, we must be able to tell the function to be used as a callback exactly what kind of event occurred and where it occurred. The function must also be told if the user was pressing such keys as Shift or Alt when the mouse event occurred. A pointer to such a function is called a `cv::MouseCallback`. This is the exact prototype that a callback function must match: `void your_mouse_callback( int event, int x, int y, int flags, void* param)` in which :

- `int event` is the event type (see Figure 1)
- `int x` is the x-location of mouse event;
- `int y` is the y-location of mouse event;
- `int flags` is a bit field in which individual bits indicate special conditions present at the time of the event (see Figure 2)
- `void* param` is a parameter from cv::setMouseCallback();

Let's see some more detailed informations about these function arguments. The first argument, called the event, will have one of the values shown in Figure 1. The second and third arguments will be set to the x and y coordinates of the mouse event. It is worth noting that these coordinates represent the pixel coordinates in the image independent of the size of the window[1]. Figure 2 shows a complete list of the flags. The final argument is a void pointer that can be used to have OpenCV pass additional information in the form of a pointer to whatever kind of structure you need[2].

[1] In general, this is not the same as the pixel coordinates of the event that would be returned by the OS. This is because OpenCV is concerned with telling you where in the image the event happened, not in the window (to which the OS typically references mouse event coordinates).

[2] A common situation in which you will want to use the param argument is when the callback itself is a static member function of a class. In this case, you will probably find yourself wanting to pass the this pointer and so indicate which class object instance the callback is intended to affect.

Figure 1: Mouse event types.

Event	Numerical value
cv::EVENT_MOUSEMOVE	0
cv::EVENT_LBUTTONDOWN	1
cv::EVENT_RBUTTONDOWN	2
cv::EVENT_MBUTTONDOWN	3
cv::EVENT_LBUTTONUP	4
cv::EVENT_RBUTTONUP	5
cv::EVENT_MBUTTONUP	6
cv::EVENT_LBUTTONDBLCLK	7
cv::EVENT_RBUTTONDBLCLK	8
cv::EVENT_MBUTTONDBLCLK	9

Figure 2: Mouse event flags.

Flag	Numerical value
cv::EVENT_FLAG_LBUTTON	1
cv::EVENT_FLAG_RBUTTON	2
cv::EVENT_FLAG_MBUTTON	4
cv::EVENT_FLAG_CTRLKEY	8
cv::EVENT_FLAG_SHIFTKEY	16
cv::EVENT_FLAG_ALTKEY	32

Next, we need the function that registers the callback. That function is called cv::setMouseCallback(const string& winname, MouseCallback onMouse, void* userdata = 0) in which:

- `winname` is the name of the window to which the callback will be attached. Only events in that particular window will trigger this specific callback and so all mouse events related to this window will be registered;
- `onMouse` is the name of the callback function;
- `userdata` allows us to specify the parameter information that should be given to the callback whenever it is executed. This is, of course, the same parameter we were just discussing with the callback prototype.

Complete the program mouse_ students.cpp to detect left, right and middle mouse clicks and mouse movements with its coordinates using 'sample.jpg' image.

2. **Drawing over an image**

We often want to draw some kind of picture, or to draw something on top of an image obtained from somewhere else. OpenCV provides easy to use functions for drawing over an image.

The drawing functions available will work with images of any depth, but most of them only affect the first three channels - defaulting to only the first channel in the case of single-channel images. Most of the drawing functions support a color, a thickness, what is called a "line type" (which really means whether or not to anti-alias lines), and subpixel alignment of objects. When specifying colors, the convention is to use the `cv::Scalar` object, even though only the first three values are used most of the time. (It is sometimes convenient to be able to use the fourth value in a `cv::Scalar` to represent an alpha-channel, but the drawing functions do not currently support alpha blending.) Also, by convention, OpenCV uses BGR ordering 27 for converting multichannel images to color renderings (this is what is used by the draw functions `imshow()`, which actually paints images onto your screen for viewing). Of course, you don't have to use this convention, and it might not be ideal if you are using data from some other library with OpenCV headers on top of it. In any case, the core functions of the library are always agnostic to any "meaning" you might assign to a channel.

In this lesson you'll study the following shapes and text which can be drawn in OpenCV:

- Line;
- Circle;
- Ellipse;
- Rectangle;
- Text.

Before seeing the functions will let you draw lines, circles, rectangles and so on, we have to introduce some OpenCV basic types: point, scalar and size.

The Point Classes

Of the OpenCV basic types, the point classes are probably the simplest. As we mentioned earlier, these are implemented based on a template structure, such that there can be points of any type: integer, floating- point, and so on. There are actually two such templates, one for two-dimensional and one for three- dimensional points. The big advantage of the point classes is that they are simple and have very little overhead. Natively, they do not have a lot of operations defined on them, but they can be cast to somewhat more generalised types, such as the fixed vector classes or the fixed matrix classes (discussed later), when needed. In most programs, the point classes are instantiated using aliases that take forms like `cv::Point2i` or `cv::Point3f`, with the last letter indicating the desired primitive from which the point is to be constructed. (Here, b is an unsigned character, s is a short integer, i is a 32-bit integer, f is a 32-bit floating-point number, and d is a 64-bit floating-point number.) Figure 3 shows the (relatively short) list of functions natively supported by the point classes.

Figure 3: Operations supported directly by the point classes.

Operation	Examples
Default constructors	`cv::Point2i p();` `cv::Point3f p();`
Copy constructor	`cv::Point3f p2( p1 );`
Value constructors	`cv::Point2i p( x0, x1 );` `cv::Point3d p( x0, x1, x2 );`
Cast to the fixed vector classes	`(cv::Vec3f) p;`

Member access	`p.x; p.y; // and for three-dimensional`
	`// point classes: p.z`

Dot product	`float x = p1.dot( p2 )`
product	
Cross product	`p1.cross( p2 ) // (for three-dimensional point` `// classes only)`
Query if point *p* is inside of rectangle *r*	`p.inside( r ) // (for two-dimensional point` `// classes only)`

class cv::Scalar

The class cv::Scalar is really a four-dimensional point class. Like the others, it is actually associated with a template class, but the alias for accessing it returns an instantiation of that template in which all of the members are double-precision floating-point numbers. The cv::Scalar class also has some special member functions associated with uses of four-component vectors in computer vision. Figure 4 sums up all the operations supported directly by this class.

Figure 4: Operations supported directly by class cv::Scalar.

Operation	Example
Default constructors	`cv::Scalar s();`
Copy constructor	`cv::Scalar s2( s1 );`
Value constructors	`cv::Scalar s( x0 );` `cv::Scalar s( x0, x1, x2, x3 );`
Element-wise multiplication	`s1.mul( s2 );`
(Quaternion) conjugation	`s.conj();  // (returns cv::Scalar(s0,-s1,-s2,-s2))`
(Quaternion) real test	`s.isReal(); // (returns true iff s1==s2==s3==0)`

The Size Classes

The size classes are, in practice, similar to the corresponding point classes, and can be cast to and from them. The primary difference between the two is that the point class'data members are named x and y, while the corresponding data members in the size classes are named width and height. The three aliases for the size classes are cv::Size, cv::Size2i, and cv::Size2f. The first two of these are equivalent and imply integer size, while the last is for 32-bit floating-point sizes. Figure 5 sums up all the operations supported directly by this class.

Figure 5: Operations supported directly by class cv::Size.

Operation	Example

Default constructors	`cv::Size sz();`
	`cv::Size2i sz();`
	`cv::Point2f sz();`

Copy constructor	`cv::Size sz2( sz1 );`
Value constructors	`cv::Size2f sz( w, h );`
Member access	`sz.width; sz.height;`
Compute area	`sz.area();`

The Rect Classes

The rectangle classes include the members x and y of the point class (representing the upper-left corner of the rectangle) and the members width and height of the size class (representing the extent of the rectangle). The rectangle classes, however, do not inherit from the point or size classes, and so in general they do not inherit operators from them.

Figure 6: Operations supported directly by class cv::Rect*

Operation	**Example**
Default constructors	`cv::Rect r();`
Copy constructor	`cv::Rect r2( r1 );`
Value constructors	`cv::Rect( x, y, w, h );`
Construct from origin and size	`cv::Rect( p, sz );`
Construct from two corners	`cv::Rect( p1, p2 );`
Member access	`r.x; r.y; r.width; r.height;`
Compute area	`r.area();`
Extract upper-left corner	`r.tl();`
Extract lower-right corner	`r.lr;`
Determine if point *p* is inside of rectangle *r*	`r.contains( p );`

Cast operators and copy constructors exist to allow cv::Rect to be computed from or cast to the old-style cv::CvRect type as well. cv::Rect is actually an alias for a rectangle template instantiated with integer members. The class cv::Rect also supports a variety of overloaded operators that can be used for the computation of various geometrical properties of two rectangles or a rectangle and another object.

Figure 7: Overloaded operators that take objects of type cv::Rect

Operation	Example
Intersection of rectangles r1 and r2	`cv::Rect r3 = r1 & r2;` `r1 &= r2;`
Minimum area rectangle containing rectangles r1 and r2	`cv::Rect r3 = r1 | r2;` `r1 |= r2;`

Translate rectangle r by an amount x	`cv::Rect rx = r + v; // v is a cv::Point2i`
	`r += v;`

Enlarge a rectangle r by an amount given by size s	`cv::Rect rs = r + s; // s is a cv::Point2i`
	`r += s;`

Compare rectangles r1 and r2 for exact equality	`bool eq = (r1 == r2);`
Compare rectangles r1 and r2 for inequality	`bool ne = (r1 != r2);`

cv::line()

`cv::line(cv::Mat& img, cv::Point pt1, cv::Point pt2, const cv::Scalar& color, int thickness = 1, int lineType, int shift = 0)` is used to draw a simple line. Here are the argument of the function in details:

- img is the image to be drawn on;
- pt1 is the first end-point of line;
- pt2 is the second end-point of line;
- color is the color of the line;
- thickness is the thickness of the line;
- lineType is the type of the line;
- shift is the number of fractional bits in the point coordinates.

The thickness of the boundary is controlled by the thickness parameter. By default, only an outline of the shape is drawn. To draw a shape that is filled with the specified color, the thickness should be -1. The lineType parameter controls the quality of rendering. When lineType is set to CV_AA, anti-aliased lines are drawn.[3]

cv::circle()

`cv::circle( cv::Mat& img, cv::Point center, int radius, const cv::Scalar& color, int thickness = 1, int lineType, int shift = 0)` is used to draw a circle. Here are the argument of the function in details:

[3]Functions that draw lines of one kind or another (segments, circles, rectangles, etc.) will usually accept a thickness and lineType parameter. Both are integers, but the only accepted values for the latter are 4, 8, or cv_AA. thickness will be the thickness of the line measured in pixels. For circles, rectangles, and all of the other closed shapes, the thickness argument can also be set to cv::FILL (which is an alias for -1). In that case, the result is that the drawn figure will be filled in the same color as the edges. The lineType argument indicates whether the lines should be "4-connected", "8-connected", or anti-aliased. For the first two, the Bresenham algorithm is used, while for the anti-aliased lines, Gaussian filtering is used. Wide lines are always drawn with rounded ends.

- img is the image to be drawn on;
- center is the location of circle center;
- radius is the radius of circle;
- color is in RGB form;
- thickness is the thickness of the line;
- lineType is the type of the line (Connectedness, 4 or 8);
- shift represents the bits of radius to treat as fraction, it is applied to both the radius and the center location.

cv::rectangle()

`cv::rectangle(cv::Mat& img, cv::Point pt1, cv::Point pt2, const cv::Scalar& color, int thickness = 1, int lineType = 8, int shift = 0` is used to draw a rectangle. Here are the argument of the function in details:

- img is the image to be drawn on;
- pt1 is the upper left corner vertex of rectangle;
- pt2 is the lower right corner vertex of rectangle;
- color is in RGB form;
- thickness is the thickness of the line;
- lineType is the type of the line (Connectedness, 4 or 8);
- shift represents the bits of radius to treat as fraction, it is applied to both the radius and the center location.

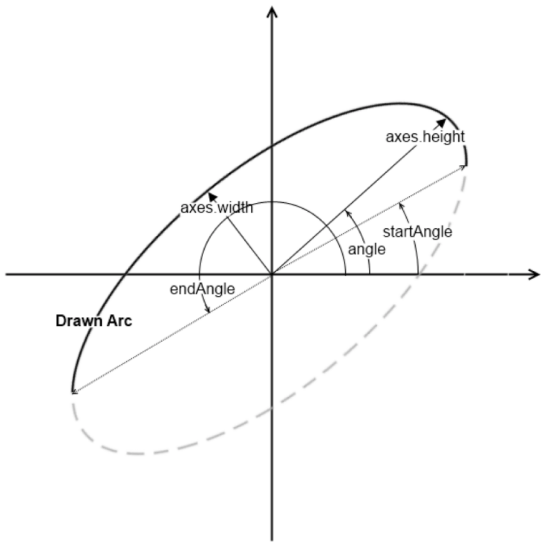
cv::ellipse()

`cv::ellipse(cv::Mat& img, cv::Point center, cv::Size axes, double angle, double startAngle, double endAngle, const cv::Scalar& color, int thickness = 1, int lineType = 8, int shift = 0` is used to draw a rectangle. Here are the argument of the function in details:

- img is the image to be drawn on;
- center is the location of ellipse center;
- axes is the length of major and minor axes;
- angle is the tilt angle of major axis;
- startAngle is the start angle for arc drawing;
- endAngle is the end angle for arc drawing;
- color is in RGB form;
- thickness is the thickness of the line;
- lineType is the type of the line (Connectedness, 4 or 8);
- shift represents the bits of radius to treat as fraction, it is applied to both the radius and the center location.

The `cv::ellipse()` function is very similar to the cv::circle() function, with the primary difference being the axes argument, which is of type `cv::Size`. In this case, the height and width arguments represent the length of the ellipse's major and minor axes. The angle is the angle (in degrees) of the major axis, which is measured counterclockwise from horizontal (i.e., from the x-axis). Similarly, the startAngle and endAngle indicate (also in degrees) the angle for the arc to start and for it to finish (see Figure 6). Thus, for a complete ellipse, you must set these values to 0 and 360, respectively.

Figure 8: An elliptical arc specified by the major and minor axes with tilt angle.



After studying and running the code 'drawShape.cpp' using 'mark.jpg' image, adapt the functions to obtain similar results on your own image portrait. Explain what you did in few lines.

3. **Drawing circles on an image**
   Complete the code 'mouse_circle_students.cpp' which draws a circle on the image. You first mark the center of the circle and then drag the mouse according to the radius desired. Multiple circles can be drawn. 'c' is used to clear the screen (the circles) and pressing ESC terminates the program.

4. **Drawing boxes**
   To sum up what you have seen in this lesson complete the program 'drawBox_students.cpp' for using a mouse to draw boxes on the screen.