Drawing Functions

Drawing functions work with matrices/images of arbitrary depth. The boundaries of the shapes can be rendered with antialiasing (implemented only for 8-bit images for now). All the functions include the parameter `color` that uses an RGB value (that may be constructed with `CV_RGB` or the **`Scalar_`** constructor) for color images and brightness for grayscale images. For color images, the channel ordering is normally *Blue, Green, Red*. This is what **`imshow()`**, **`imread()`**, and **`imwrite()`** expect. So, if you form a color using the `Scalar` constructor, it should look like:

$$\texttt{Scalar}(\texttt{blue\_component}, \texttt{green\_component}, \texttt{red\_component}[, \texttt{alpha\_component}])$$

If you are using your own image rendering and I/O functions, you can use any channel ordering. The drawing functions process each channel independently and do not depend on the channel order or even on the used color space. The whole image can be converted from BGR to RGB or to a different color space using **`cvtColor()`** .

If a drawn figure is partially or completely outside the image, the drawing functions clip it. Also, many drawing functions can handle pixel coordinates specified with sub-pixel accuracy. This means that the coordinates can be passed as fixed-point numbers encoded as integers. The number of fractional bits is specified by the `shift` parameter and the real point coordinates are calculated as $\texttt{Point}(x, y) \rightarrow \texttt{Point2f}(x * 2^{-\texttt{shift}}, y * 2^{-\texttt{shift}})$. This feature is especially effective when rendering antialiased shapes.

> Note: The functions do not support alpha-transparency when the target image is 4-channel. In this case, the `color[3]` is simply copied to the repainted pixels. Thus, if you want to paint semi-transparent shapes, you can paint them in a separate buffer and then blend it with the main image.

> Note:
>
> - An example on using variate drawing functions like line, rectangle, ... can be found at opencv_source_code/samples/cpp/drawing.cpp

circle

Draws a circle.

C++: void **`circle`**(Mat& img, Point center, int radius, const Scalar& color, int thickness=1, int lineType=8, int shift=0)

Python: `cv2.`**`circle`**(img, center, radius, color[, thickness[, lineType[, shift]]]) →

None

C: void **cvCircle**(CvArr* img, CvPoint center, int radius, CvScalar color, int thickness=1, int line_type=8, int shift=0)

Python: `cv.`**Circle**(img, center, radius, color, thickness=1, lineType=8, shift=0) → None

Parameters:
- img – Image where the circle is drawn.
- center – Center of the circle.
- radius – Radius of the circle.
- color – Circle color.
- thickness – Thickness of the circle outline, if positive. Negative thickness means that a filled circle is to be drawn.
- lineType – Type of the circle boundary. See the **line()** description.
- shift – Number of fractional bits in the coordinates of the center and in the radius value.

The function `circle` draws a simple or filled circle with a given center and radius.

clipLine

Clips the line against the image rectangle.

C++: bool **clipLine**(Size imgSize, Point& pt1, Point& pt2)

C++: bool **clipLine**(Rect imgRect, Point& pt1, Point& pt2)

Python: `cv2.`**clipLine**(imgRect, pt1, pt2) → retval, pt1, pt2

C: int **cvClipLine**(CvSize img_size, CvPoint* pt1, CvPoint* pt2)

Python: `cv.`**ClipLine**(imgSize, pt1, pt2) -> (point1, point2)

Parameters:
- imgSize – Image size. The image rectangle is `Rect(0, 0, imgSize.width, imgSize.height)` .
- imgRect – Image rectangle.
- pt1 – First line point.
- pt2 – Second line point.

The functions `clipLine` calculate a part of the line segment that is entirely within the specified rectangle. They return `false` if the line segment is completely outside the rectangle. Otherwise, they return `true` .

ellipse

Draws a simple or thick elliptic arc or fills an ellipse sector.

C++: void **ellipse**(Mat& img, Point center, Size axes, double angle, double startAngle, double endAngle, const Scalar& color, int thickness=1, int lineType=8, int shift=0)

C++: void **ellipse**(Mat& img, const RotatedRect& box, const Scalar& color, int thickness=1, int lineType=8)

Python: `cv2.`**ellipse**(img, center, axes, angle, startAngle, endAngle, color[, thickness[, lineType[, shift]]]) → None

Python: `cv2.`**ellipse**(img, box, color[, thickness[, lineType]]) → None

C: void **cvEllipse**(CvArr* img, CvPoint center, CvSize axes, double angle, double start_angle, double end_angle, CvScalar color, int thickness=1, int line_type=8, int shift=0)

Python: `cv.`**Ellipse**(img, center, axes, angle, start_angle, end_angle, color, thickness=1, lineType=8, shift=0) → None

C: void **cvEllipseBox**(CvArr* img, CvBox2D box, CvScalar color, int thickness=1, int line_type=8, int shift=0)

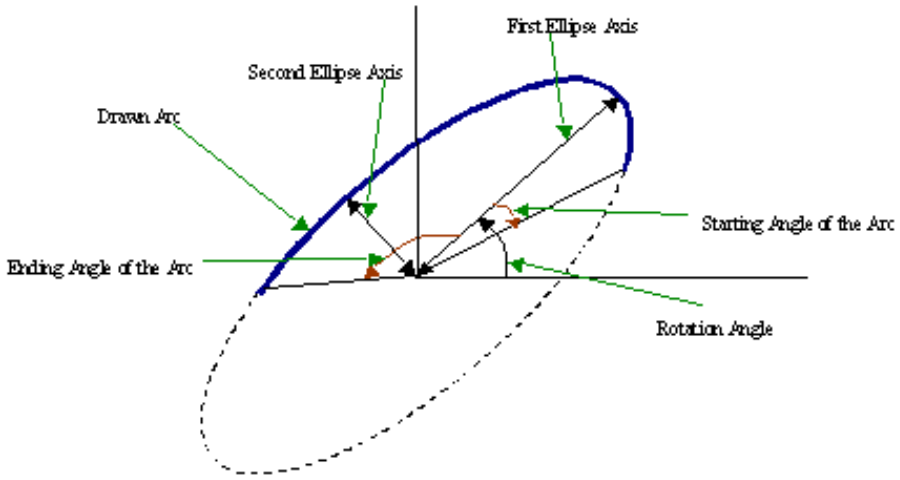
Python: `cv.`**EllipseBox**(img, box, color, thickness=1, lineType=8, shift=0) → None

Parameters:
- img – Image.
- center – Center of the ellipse.
- axes – Half of the size of the ellipse main axes.
- angle – Ellipse rotation angle in degrees.
- startAngle – Starting angle of the elliptic arc in degrees.
- endAngle – Ending angle of the elliptic arc in degrees.
- box – Alternative ellipse representation via **RotatedRect** or `CvBox2D`. This means that the function draws an ellipse inscribed in the rotated rectangle.
- color – Ellipse color.
- thickness – Thickness of the ellipse arc outline, if positive. Otherwise, this indicates that a filled ellipse sector is to be drawn.
- lineType – Type of the ellipse boundary. See the **line()** description.
- shift – Number of fractional bits in the coordinates of the center and values of axes.

The functions `ellipse` with less parameters draw an ellipse outline, a filled ellipse, an elliptic arc, or a filled ellipse sector. A piecewise-linear curve is used to approximate the elliptic arc boundary. If you need more control of the ellipse rendering, you can retrieve the curve using **ellipse2Poly()** and then render it with **polylines()** or fill it

with **`fillPoly()`** . If you use the first variant of the function and want to draw the whole ellipse, not an arc, pass `startAngle=0` and `endAngle=360` . The figure below explains the meaning of the parameters.

Figure 1. Parameters of Elliptic Arc



ellipse2Poly

Approximates an elliptic arc with a polyline.

C++: void **`ellipse2Poly`**(Point center, Size axes, int angle, int arcStart, int arcEnd, int delta, vector<Point>& pts)

Python: `cv2.`**`ellipse2Poly`**(center, axes, angle, arcStart, arcEnd, delta) → pts

Parameters:
- center – Center of the arc.
- axes – Half of the size of the ellipse main axes. See the **`ellipse()`** for details.
- angle – Rotation angle of the ellipse in degrees. See the **`ellipse()`** for details.
- arcStart – Starting angle of the elliptic arc in degrees.
- arcEnd – Ending angle of the elliptic arc in degrees.
- delta – Angle between the subsequent polyline vertices. It defines the approximation accuracy.
- pts – Output vector of polyline vertices.

The function `ellipse2Poly` computes the vertices of a polyline that approximates the specified elliptic arc. It is used by **`ellipse()`** .

fillConvexPoly

Fills a convex polygon.

C++: void **fillConvexPoly**(Mat& img, const Point* pts, int npts, const Scalar& color, int lineType=8, int shift=0)

Python: `cv2.` **fillConvexPoly**(img, points, color[, lineType[, shift]]) → None

C: void **cvFillConvexPoly**(CvArr* img, const CvPoint* pts, int npts, CvScalar color, int line_type=8, int shift=0)

Python: `cv.` **FillConvexPoly**(img, pn, color, lineType=8, shift=0) → None

Parameters:
- img – Image.
- pts – Polygon vertices.
- npts – Number of polygon vertices.
- color – Polygon color.
- lineType – Type of the polygon boundaries. See the **line()** description.
- shift – Number of fractional bits in the vertex coordinates.

The function `fillConvexPoly` draws a filled convex polygon. This function is much faster than the function `fillPoly` . It can fill not only convex polygons but any monotonic polygon without self-intersections, that is, a polygon whose contour intersects every horizontal line (scan line) twice at the most (though, its top-most and/or the bottom edge could be horizontal).

fillPoly

Fills the area bounded by one or more polygons.

C++: void **fillPoly**(Mat& img, const Point** pts, const int* npts, int ncontours, const Scalar& color, int lineType=8, int shift=0, Point offset=Point())

Python: `cv2.` **fillPoly**(img, pts, color[, lineType[, shift[, offset]]]) → None

C: void **cvFillPoly**(CvArr* img, CvPoint** pts, const int* npts, int contours, CvScalar color, int line_type=8, int shift=0)

Python: `cv.` **FillPoly**(img, polys, color, lineType=8, shift=0) → None

Parameters:
- img – Image.
- pts – Array of polygons where each polygon is represented as an array of points.
- npts – Array of polygon vertex counters.
- ncontours – Number of contours that bind the filled region.
- color – Polygon color.

- lineType – Type of the polygon boundaries. See the **line()** description.
- shift – Number of fractional bits in the vertex coordinates.
- offset – Optional offset of all points of the contours.

The function `fillPoly` fills an area bounded by several polygonal contours. The function can fill complex areas, for example, areas with holes, contours with self-intersections (some of their parts), and so forth.

getTextSize

Calculates the width and height of a text string.

C++: Size **getTextSize**(const string& text, int fontFace, double fontScale, int thickness, int* baseLine)

Python: `cv2.`**getTextSize**(text, fontFace, fontScale, thickness) → retval, baseLine

C: void **cvGetTextSize**(const char* text_string, const CvFont* font, CvSize* text_size, int* baseline)

Python: `cv.`**GetTextSize**(textString, font)-> (textSize, baseline)

Parameters:
- text – Input text string.
- text_string – Input text string in C format.
- fontFace – Font to use. See the **putText()** for details.
- fontScale – Font scale. See the **putText()** for details.
- thickness – Thickness of lines used to render the text. See **putText()** for details.
- baseLine – Output parameter - y-coordinate of the baseline relative to the bottom-most text point.
- baseline – Output parameter - y-coordinate of the baseline relative to the bottom-most text point.
- font – Font description in terms of old C API.
- text_size – Output parameter - The size of a box that contains the specified text.

The function `getTextSize` calculates and returns the size of a box that contains the specified text. That is, the following code renders some text, the tight box surrounding it, and the baseline:

```
string text = "Funny text inside the box";
int fontFace = FONT_HERSHEY_SCRIPT_SIMPLEX;
double fontScale = 2;
int thickness = 3;

Mat img(600, 800, CV_8UC3, Scalar::all(0));
```

```
int baseline=0;
Size textSize = getTextSize(text, fontFace,
                            fontScale, thickness, &baseline);
baseline += thickness;

// center the text
Point textOrg((img.cols - textSize.width)/2,
              (img.rows + textSize.height)/2);

// draw the box
rectangle(img, textOrg + Point(0, baseline),
          textOrg + Point(textSize.width, -textSize.height),
          Scalar(0,0,255));
// ... and the baseline first
line(img, textOrg + Point(0, thickness),
     textOrg + Point(textSize.width, thickness),
     Scalar(0, 0, 255));

// then put the text itself
putText(img, text, textOrg, fontFace, fontScale,
        Scalar::all(255), thickness, 8);
```

InitFont

Initializes font structure (OpenCV 1.x API).

C: void **cvInitFont**(CvFont* font, int font_face, double hscale, double vscale, double shear=0, int thickness=1, int line_type=8)

Parameters:
- font – Pointer to the font structure initialized by the function
- font_face –

  Font name identifier. Only a subset of Hershey fonts http://sources.isc.org/utils/misc/hershey-font.txt are supported now:

  - CV_FONT_HERSHEY_SIMPLEX normal size sans-serif font
  - CV_FONT_HERSHEY_PLAIN small size sans-serif font
  - CV_FONT_HERSHEY_DUPLEX normal size sans-serif font (more complex than `CV_FONT_HERSHEY_SIMPLEX`)
  - CV_FONT_HERSHEY_COMPLEX normal size serif font
  - CV_FONT_HERSHEY_TRIPLEX normal size serif font (more complex than `CV_FONT_HERSHEY_COMPLEX`)
  - CV_FONT_HERSHEY_COMPLEX_SMALL smaller version of `CV_FONT_HERSHEY_COMPLEX`
  - CV_FONT_HERSHEY_SCRIPT_SIMPLEX hand-writing style font
  - CV_FONT_HERSHEY_SCRIPT_COMPLEX more complex

variant of `CV_FONT_HERSHEY_SCRIPT_SIMPLEX`

The parameter can be composited from one of the values above and an optional `CV_FONT_ITALIC` flag, which indicates italic or oblique font.

- hscale – Horizontal scale. If equal to `1.0f` , the characters have the original width depending on the font type. If equal to `0.5f` , the characters are of half the original width.
- vscale – Vertical scale. If equal to `1.0f` , the characters have the original height depending on the font type. If equal to `0.5f` , the characters are of half the original height.
- shear – Approximate tangent of the character slope relative to the vertical line. A zero value means a non-italic font, `1.0f` means about a 45 degree slope, etc.
- thickness – Thickness of the text strokes
- line_type – Type of the strokes, see **`line()`** description

The function initializes the font structure that can be passed to text rendering functions.

See also: