Technological Institute of the Philippines	**Quezon City - Computer Engineering**
Course Code:	CPE 313
Code Title:	Advanced Machine Learning and Deep Learning
1st Semester	AY 2024-2025
ACTIVITY NO. 4	**Edge and Contour Detection**
Name	De los Reyes, Jann Moises Nyll
Section	CPE32S3
Date Performed:	February 21, 2025
Date Submitted:	February 21, 2025
Instructor:	Engr. Roman M. Richard

1. Objectives

This activity aims tointroduce students to the use of OpenCV for edge detection and contour detection.

2. Intended Learning Outcomes (ILOs)

After this activity, the students should be able to:

- Explain fundamental idea of convolution and the kernel's importance.
- Use different image manipulation methods by using openCV functions.
- Perform contour and edge line drawing on their own images.

3. Procedures and Outputs

In [1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [2]:
```
import numpy as np
import cv2

# This is the file URL: https://media.wired.com/photos/5cdefb92b86e041493d389df/4:3/w_1330,h_998,c_limit/Cul
src = cv2.imread('/content/drive/MyDrive/CVdata/grumpy-cat.png')
dst = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)
```

Edges play a major role in both human and computer vision. We, as humans, can easily recognize many object types and their pose just by seeing a backlit silhouette or a rough sketch. Indeed, when art emphasizes edges and poses, it often seems to convey the idea of an archetype, such as Rodin's The Thinker or Joe Shuster's Superman.

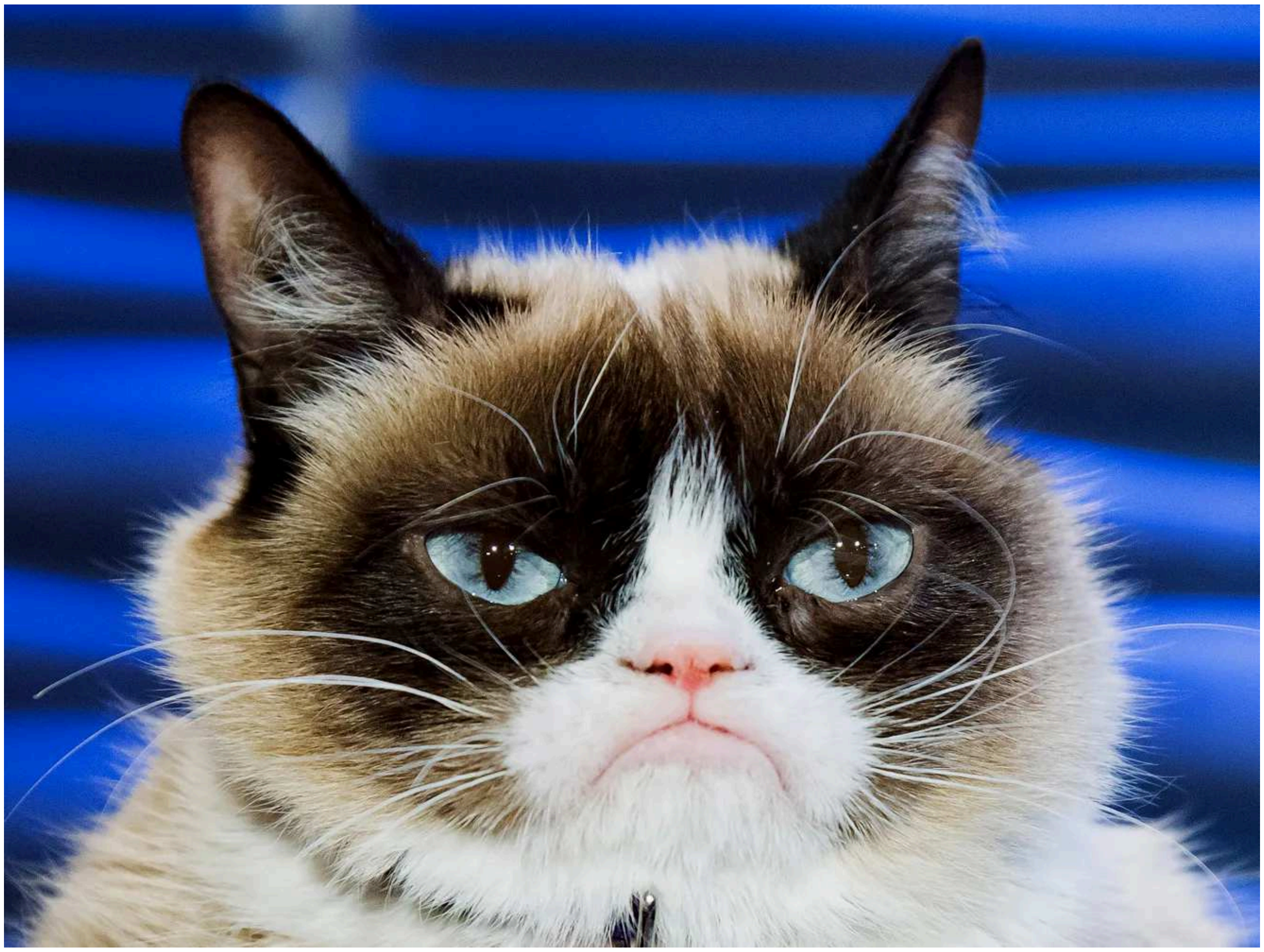
Software, too, can reason about edges, poses, and archetypes.

OpenCV provides many edge-finding filters, including Laplacian(), Sobel(), and Scharr(). These filters are supposed to turn non-edge regions to black while turning edge regions to white or saturated colors. However, they are prone to misidentifying noise as edges. This flaw can be mitigated by blurring an image before trying to find its edges. OpenCV also provides many blurring filters, including blur() (simple average), medianBlur(), and GaussianBlur(). The arguments for the edge-finding and blurring filters vary but always include ksize, an odd whole number that represents the width and height (in pixels) of a filter's kernel.

For blurring, let's use medianBlur(), which is effective in removing digital video noise, especially in color images. For edge-finding, let's use Laplacian(), which produces bold edge lines, especially in grayscale images. After applying medianBlur(), but before applying Laplacian(), we should convert the image from BGR to grayscale.

Once we have the result of Laplacian(), we can invert it to get black edges on a white background. Then, we can normalize it (so that its values range from 0 to 1) and multiply it with the source image to darken the edges. Let's implement this approach:

In [3]:
```
from google.colab.patches import cv2_imshow

cv2_imshow(src)
```



In [4]:
```
cv2_imshow(dst)
```

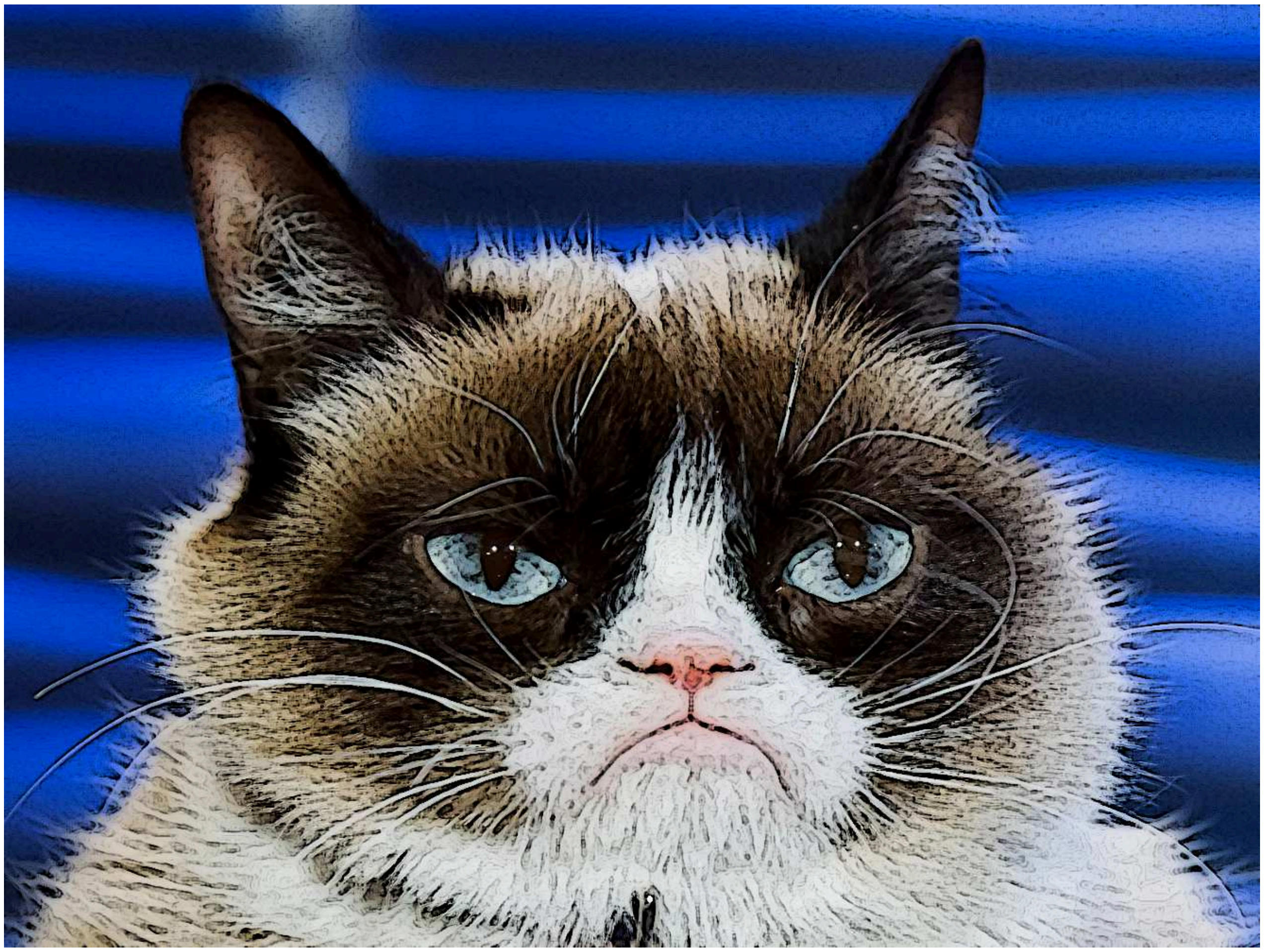
In [5]:
```
def strokeEdges(src, dst, blurKsize = 7, edgeKsize = 5):
  if blurKsize >= 3:
    blurredSrc = cv2.medianBlur(src, blurKsize)
    graySrc = cv2.cvtColor(blurredSrc, cv2.COLOR_BGR2GRAY)
  else:
    graySrc = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)

  cv2.Laplacian(graySrc, cv2.CV_8U, graySrc, ksize = edgeKsize)
  normalizedInverseAlpha = (1.0 / 255) * (255 - graySrc)
  channels = cv2.split(src)

  for channel in channels:
    channel[:] = channel * normalizedInverseAlpha

  return cv2.merge(channels,dst)
```

For the function above, provide an analysis:

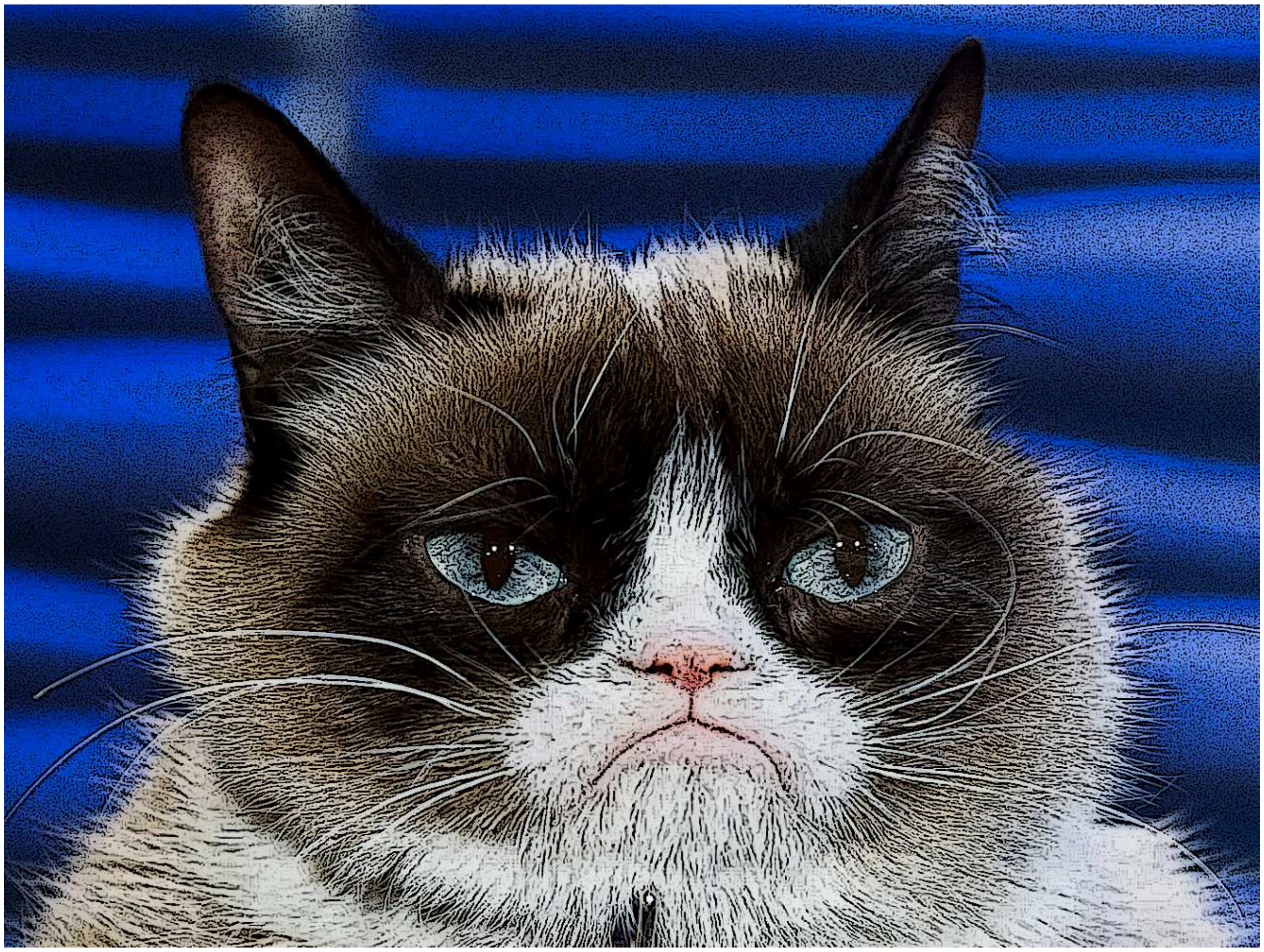
- Try to run the function and pass values for its parameters.
- Change the values in the kSize variables. What do you notice?

Make sure to add codeblocks underneath this section to ensure that your demonstration of the procedure and answers are easily identifiable

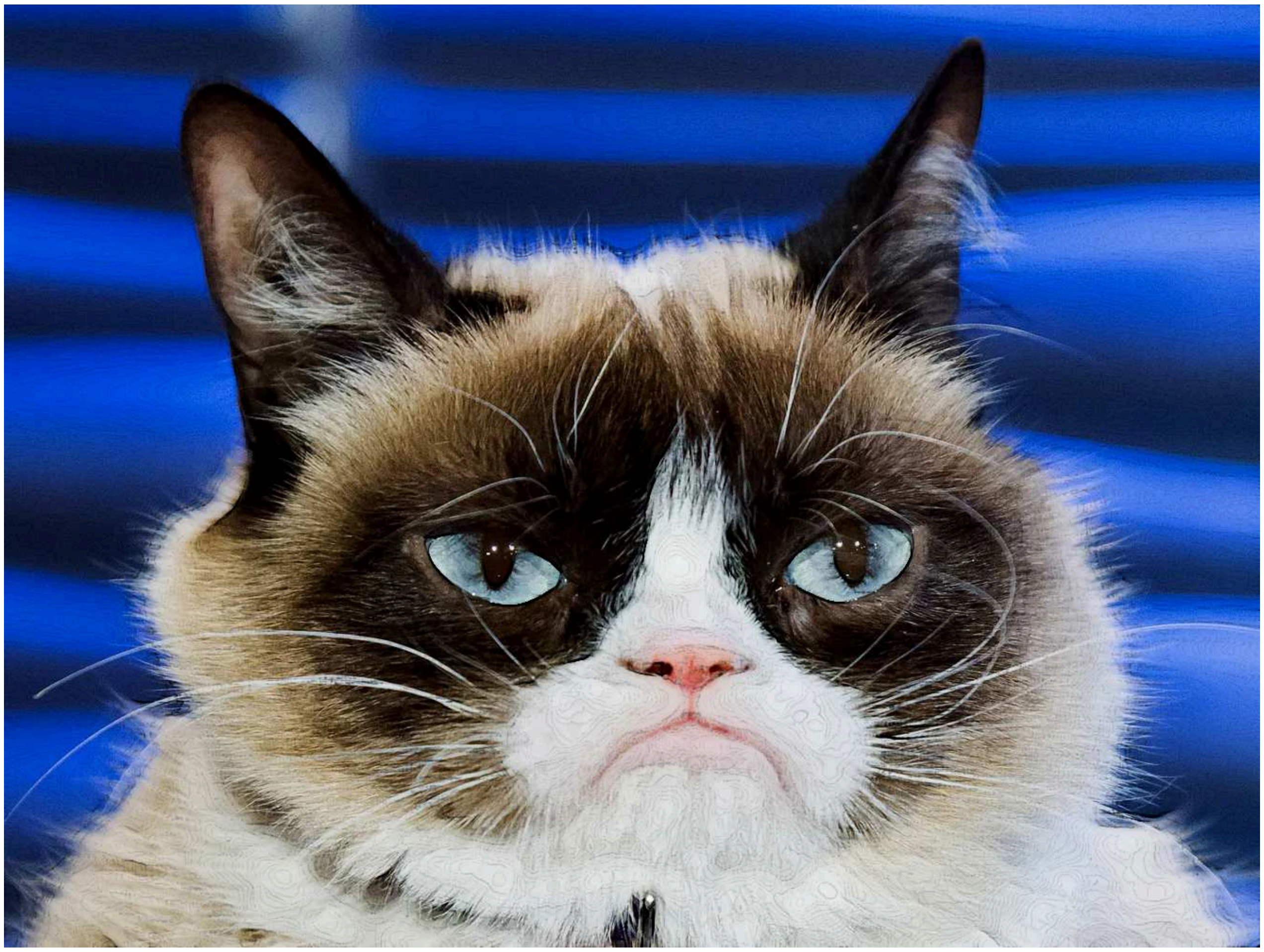
In [6]:
```
# Sample

new_img = strokeEdges(src, dst)
cv2_imshow(new_img)
```



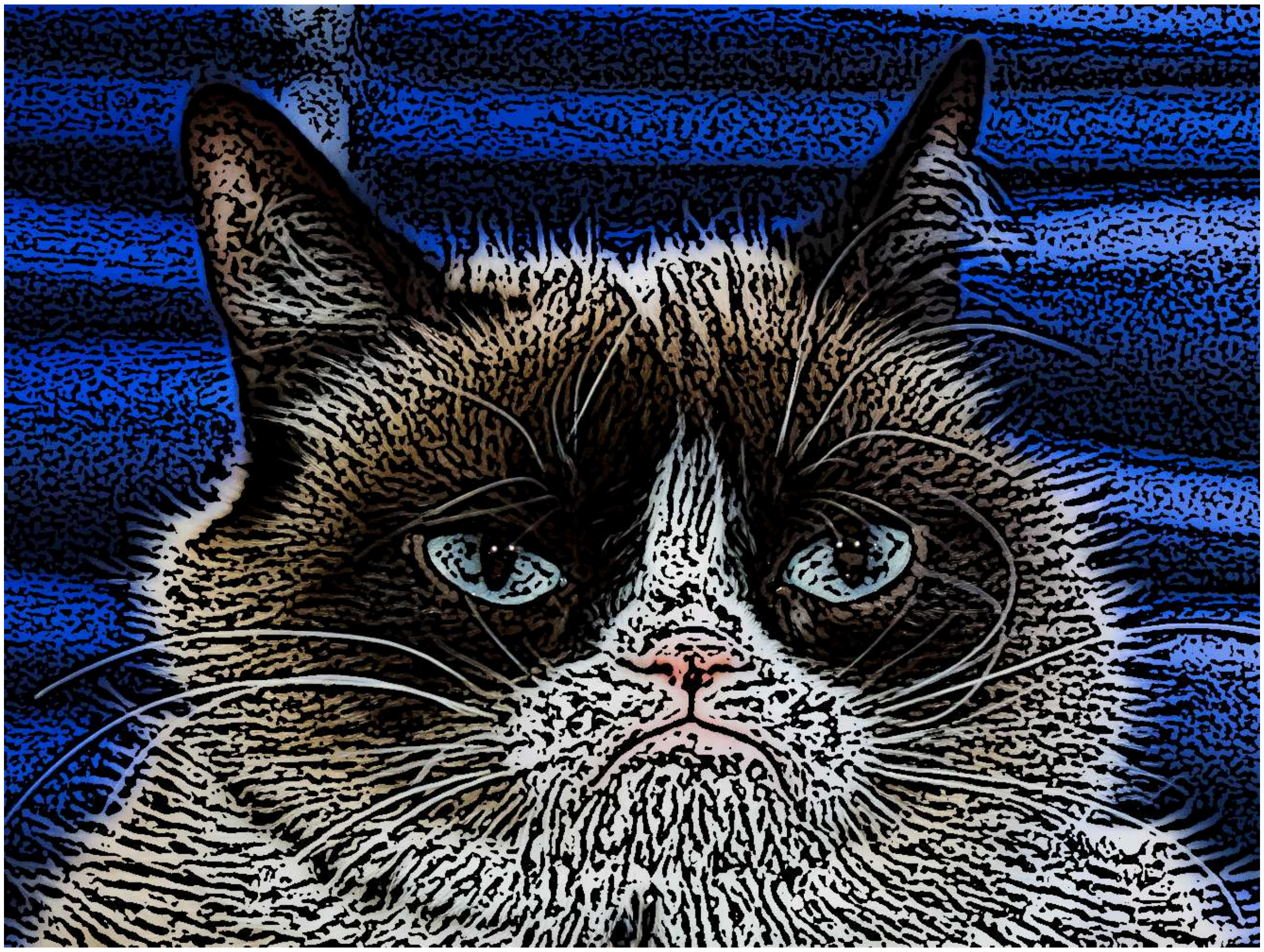
In [7]:
```
#Sample B

B_img = strokeEdges(src,dst,blurKsize= 2)
cv2_imshow(B_img)
```



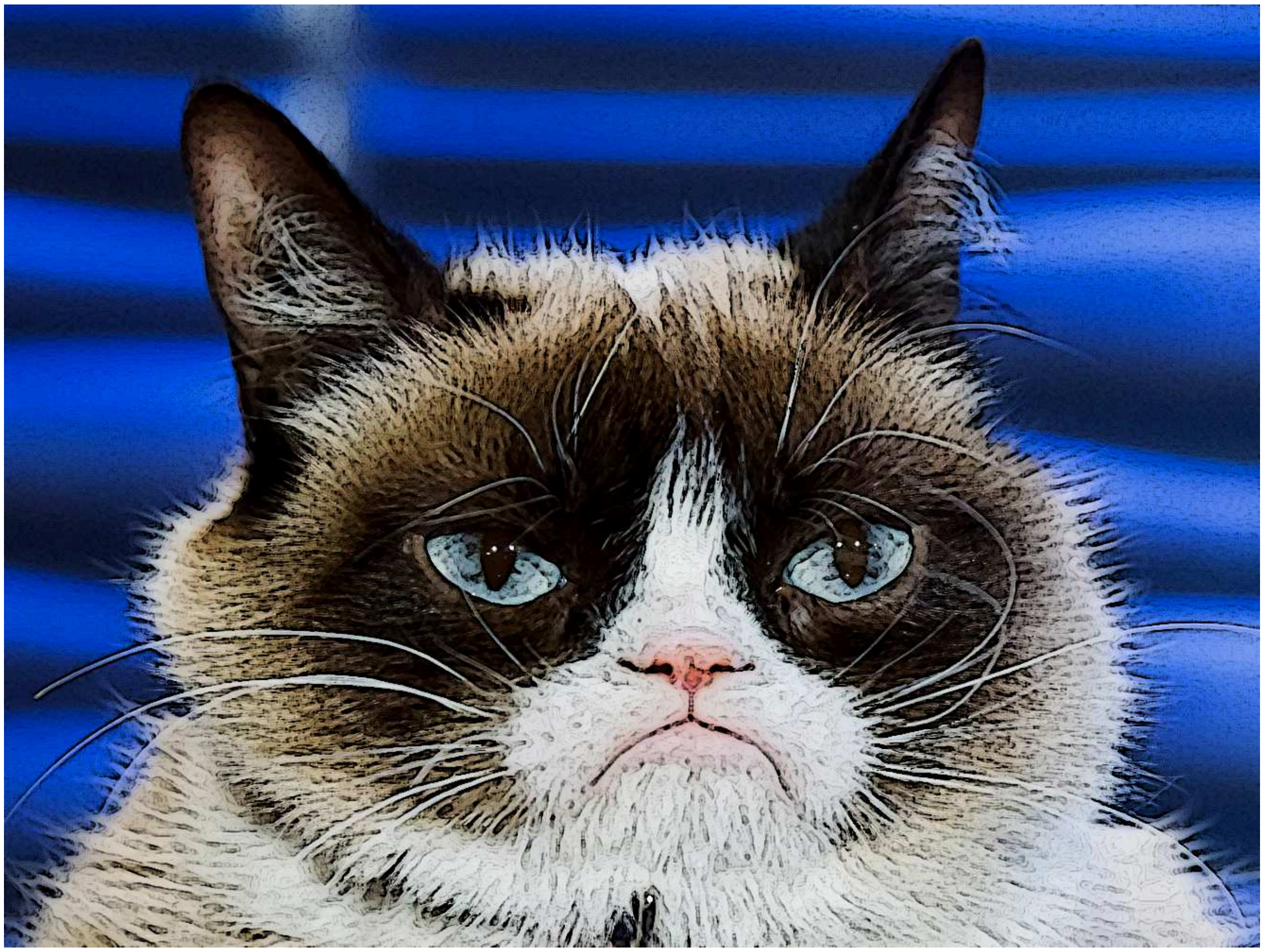
In [8]:
```
#sample C

C_img = strokeEdges(src,dst,blurKsize=25)
cv2_imshow(C_img)
```



In [9]:
```
#Sample D

D_img = strokeEdges(src,dst,edgeKsize=15)
cv2_imshow(D_img)
```



In [10]:
```
#Sample E

E_img = strokeEdges(src,dst,edgeKsize=5)
cv2_imshow(E_img)
```



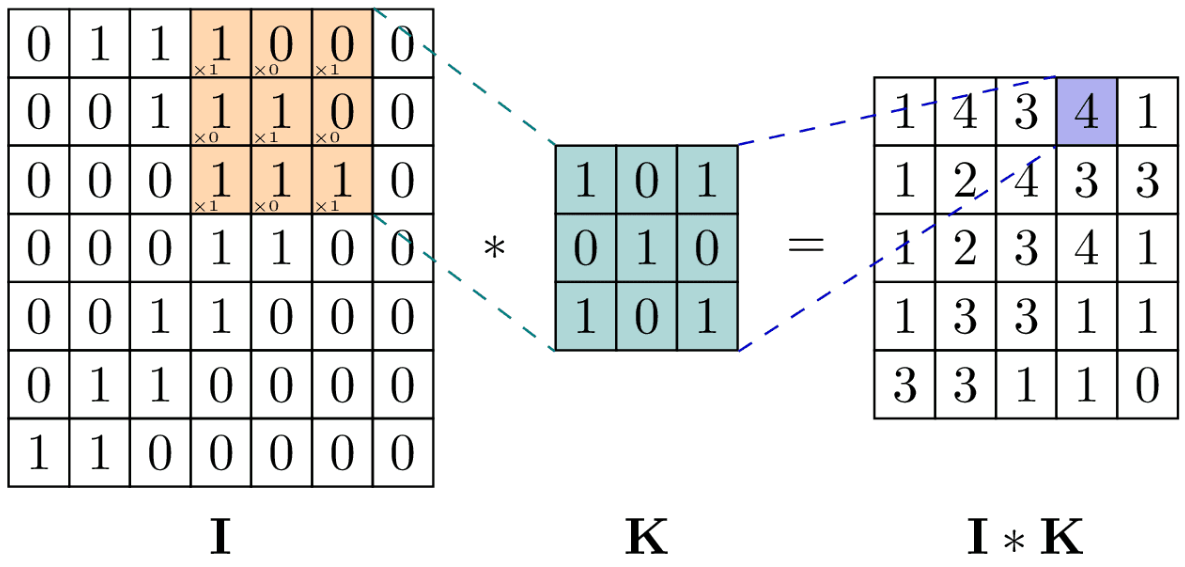
Observation:

Based on the following sample testing of various Ksize values above, we notice that theres a change in each parameters. The blurKsize determine the amount of blurring applied in the image while edgeKsize control the size of our edge detection. In blurKsize the higher the value, the edge will appear more smoother since it increase its blurring; decreasing it on the other hand will lead no more detailed and possible noisy edge to be detected.

Increasing the edgeKsize will make our edges thicker while decreasing it make our edges to become thiner and finer.

Custom Kernels -- Getting Convoluted

As we have just seen, many of OpenCV's predefined filters use a kernel. Remember that a kernel is a set of weights, which determine how each output pixel is calculated from a neighborhood of input pixels. Another term for a kernel is a convolution matrix. It mixes up or convolves the pixels in a region. Similarly, a kernel-based filter may be called a convolution filter.



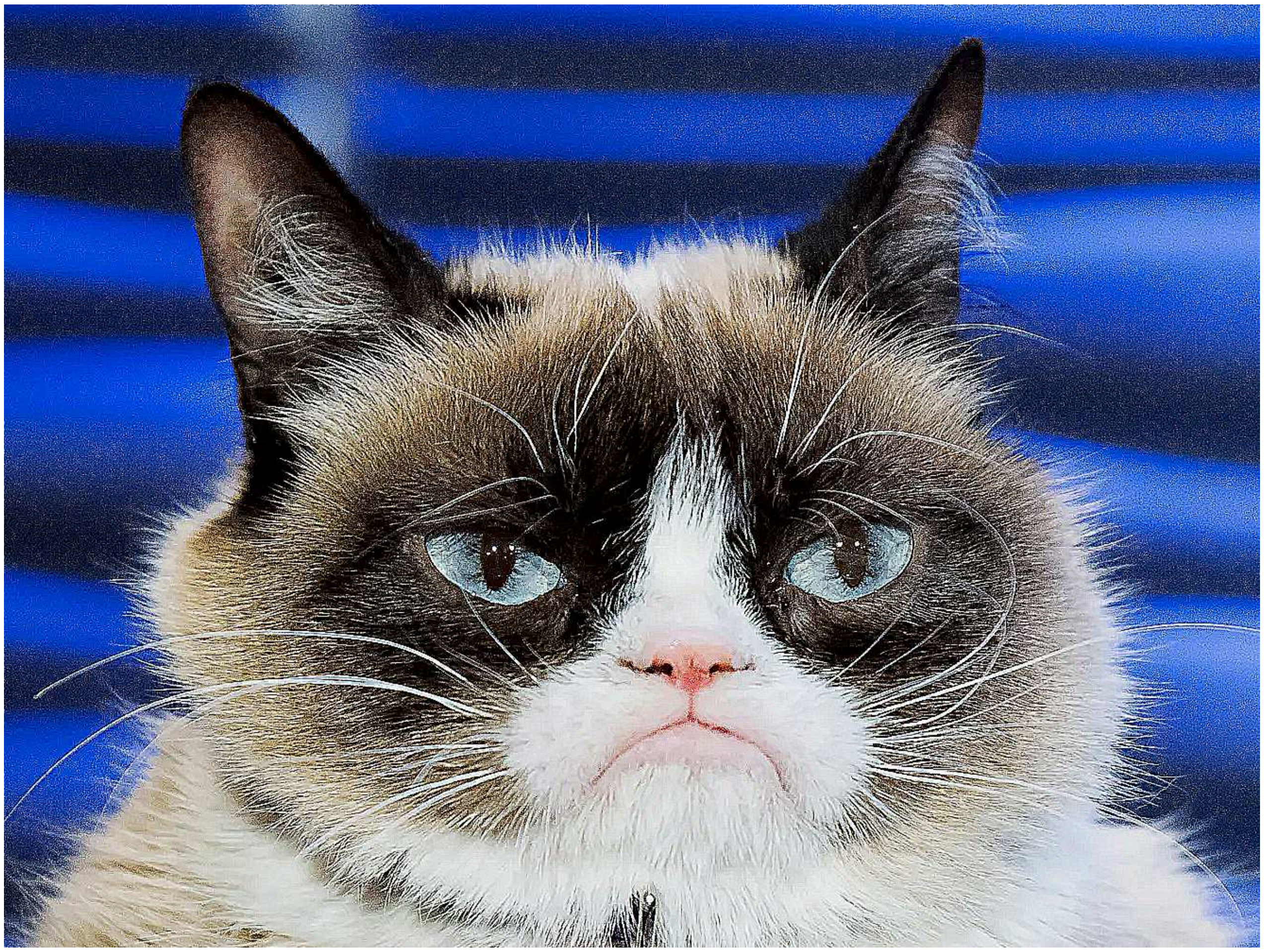
OpenCV provides a very versatile filter2D() function, which applies any kernel or convolution matrix that we specify. To understand how to use this function, let's first learn the format of a convolution matrix. It is a 2D array with an odd number of rows and columns. The central element corresponds to a pixel of interest and the other elements correspond to the neighbors of this pixel. Each element contains an integer or floating point value, which is a weight that gets applied to an input pixel's value.

In [11]:
```
# Example

kernel = np.array([[-1, -1, -1],
                   [-1, 9, -1],
                   [-1, -1, -1]])
```

Here, the pixel of interest has a weight of 9 and its immediate neighbors each have a weight of -1. For the pixel of interest, the output color will be nine times its input color minus the input colors of all eight adjacent pixels. If the pixel of interest is already a bit different from its neighbors, this difference becomes intensified. The effect is that the image looks sharper as the contrast between the neighbors is increased.

In [12]:
```
catpath = '/content/drive/MyDrive/CVdata/grumpy-cat.png'
```

In [13]:
```
# Applying the Kernel

# Reloading src and dst
src = cv2.imread(catpath)
dst = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)

# Using the filter with our Kernel
new_img2 = cv2.filter2D(src, -1, kernel, dst)

# Display
cv2_imshow(new_img2)
```



Answer the following with analysis:

- What does filter2d() function do that resulted to this image above?
- Provide a comparison between this new image and the previous image that we were able to generate.

The filter2d() function is used to apply a filter to an image. filter2d() works by convolving the filter kernel with the image. This mean that the kernel get through the image and for each pixel, the kernel is multiplied by correspoding pixel in the the image which will be summed. The sum we got will be assigned to the current pixel output of an image.

Based on the result, the `new_img2` is is more sharpened and got a little blurring comapre to the `src` image. We can also see that the output of the new images has a present of noise, thinner edges and slightly blurred look.

Based on this simple example, let's add two classes. One class, VConvolutionFilter, will represent a convolution filter in general. A subclass, SharpenFilter, will represent our sharpening filter specifically.

In [14]:
```
class VConvolutionFilter(object):
  """a filter that applies a convolution to V (or all of BGR)."""

  def __init__(self, kernel):
    self._kernel = kernel

  def apply(self, src, dst):
    """ Apply the filter with a BGR or gray source/destination """
    cv2.filter2D(src, -1, self._kernel, dst)
```

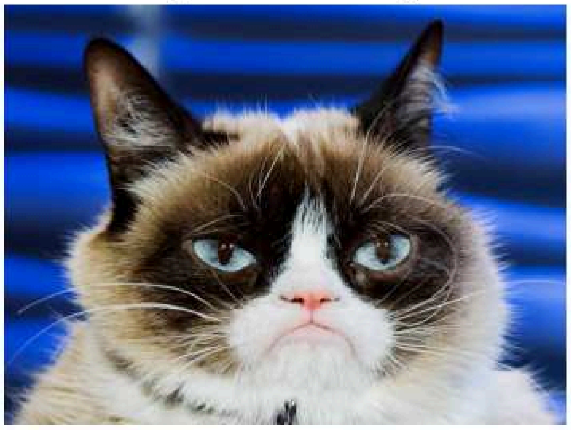
In [15]:
```
class SharpenFilter(VConvolutionFilter):
  """a sharpen filter with a 1-pixel radius."""

  def __init__(self):
    kernel = np.array([[-1, -1, -1],
                       [-1, 9, -1],
                       [-1, -1, -1]])
    VConvolutionFilter.__init__(self, kernel)
```

Run the classes above, create objects and aim to show the output of the two classes. Afterwards, make sure to make an analysis of the output.

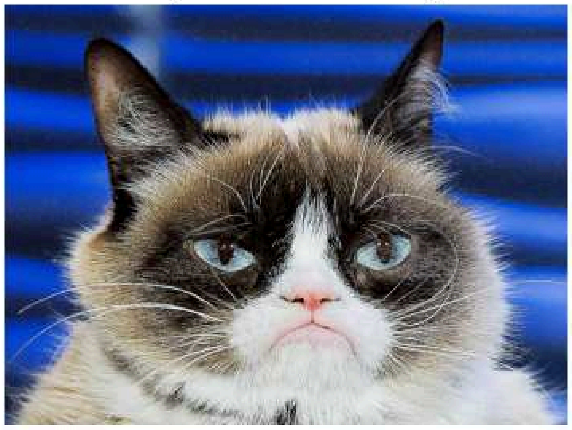
In [26]:
```
# Reloading src and dst
src = cv2.imread(catpath)
dst = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)
```

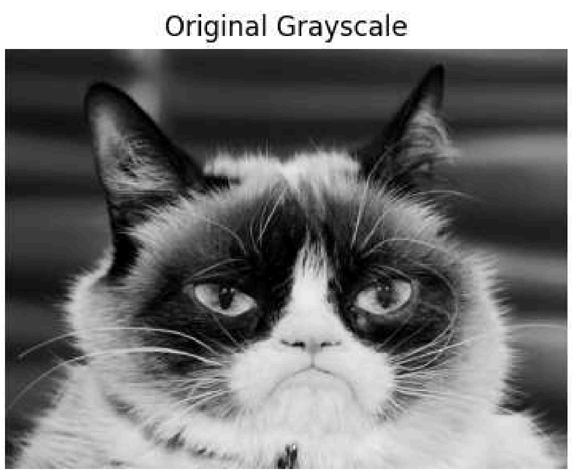
In [31]:
```
import matplotlib.pyplot as plt

# Apply the sharpened image
sharpen_filter = SharpenFilter()
dst_sharpened_color = np.zeros_like(src)
sharpen_filter.apply(src, dst_sharpened_color)

# Apply grayscale
src_gray = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY) #convert the src to grayscale
dst_sharpened = np.zeros_like(src_gray)
sharpen_filter.apply(src_gray, dst_sharpened)

# Display the original and sharpened colored images
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(cv2.cvtColor(src, cv2.COLOR_BGR2RGB))
axes[0].set_title("Original Colored Image")
axes[0].axis("off")

axes[1].imshow(cv2.cvtColor(dst_sharpened_color, cv2.COLOR_BGR2RGB))
axes[1].set_title("Sharpened Colored Image")
axes[1].axis("off")

# Display the original and sharpened grayscale images
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(src_gray, cmap="gray")
axes[0].set_title("Original Grayscale")
axes[0].axis("off")

axes[1].imshow(dst_sharpened, cmap="gray")
axes[1].set_title("Sharpened Grayscale")
axes[1].axis("off")

plt.show()
```

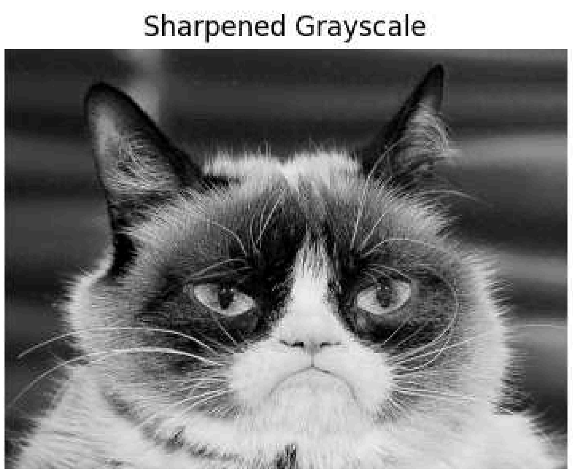
Original Colored Image



Sharpened Colored Image







Analysis

The original image is more natural than the shapened image. The sharpened image enhance the edges of our images and produce noises.

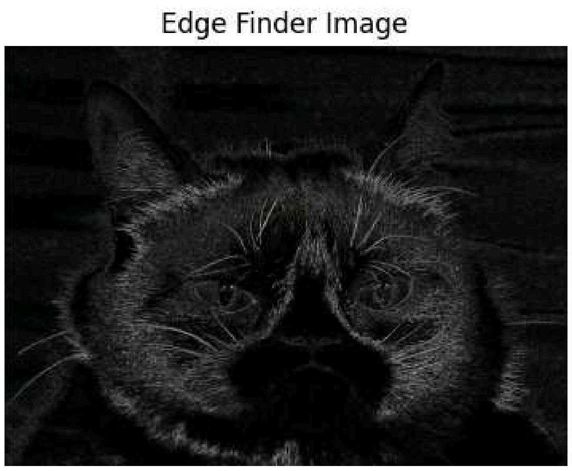
Note that the weights sum up to 1. This should be the case whenever we want to leave the image's overall brightness unchanged. If we modify a sharpening kernel slightly so that its weights sum up to 0 instead, we have an edge detection kernel that turns edges white and non-edges black.

In [34]:
```
class FindEdgesFilter(VConvolutionFilter):
  """An edge-finding filter with a 1-pixel radius."""

  def __init__(self):
    kernel = np.array([[-1, -1, -1],
                       [-1, 8, -1],
                       [-1, -1, -1]])
    VConvolutionFilter.__init__(self, kernel)
```

Run this class and demonstrate the output. Provide an analysis

In [42]:
```
findedges_filter = FindEdgesFilter()
dst_findedge_color = np.zeros_like(src)
findedges_filter.apply(src, dst_findedge_color)

# Display the original and Edge Finder images
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(cv2.cvtColor(src, cv2.COLOR_BGR2RGB))
axes[0].set_title("Original Colored Image")
axes[0].axis("off")

axes[1].imshow(cv2.cvtColor(dst_findedge_color, cv2.COLOR_BGR2RGB))
axes[1].set_title("Edge Finder Image")
axes[1].axis("off")
```

Out[42]:
```
(-0.5, 1329.5, 997.5, -0.5)
```



Analysis

The output image shows where are the edges located within a 1 pixel radius. The white color of the output image indicate the values of edges in each pixel.

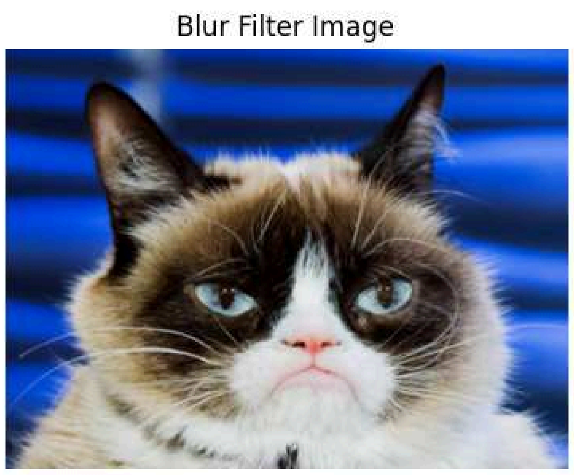
Next, let's make a blur filter. Generally, for a blur effect, the weights should sum up to 1 and should be positive throughout the neighborhood. For example, we can take a simple average of the neighborhood as follows

In [43]:
```
class Blurfilter(VConvolutionFilter):
  """A blur filter with a 2-pixel radius"""

  def __init__(self):
    kernel = np.array([[0.04, 0.04, 0.04, 0.04, 0.04],
                       [0.04, 0.04, 0.04, 0.04, 0.04],
                       [0.04, 0.04, 0.04, 0.04, 0.04],
                       [0.04, 0.04, 0.04, 0.04, 0.04],
                       [0.04, 0.04, 0.04, 0.04, 0.04]])
    VConvolutionFilter.__init__(self, kernel)
```

Run this class and demonstrate the output. Provide an analysis

In [51]:
```
blur_filter = Blurfilter()
dst_blur_color = np.zeros_like(src)
blur_filter.apply(src, dst_blur_color)

# Display the original and blur filter images
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(cv2.cvtColor(src, cv2.COLOR_BGR2RGB))
axes[0].set_title("Original Colored Image")
axes[0].axis("off")

axes[1].imshow(cv2.cvtColor(dst_blur_color, cv2.COLOR_BGR2RGB))
axes[1].set_title("Blur Filter Image")
axes[1].axis("off")
```

Out[51]:
```
(-0.5, 1329.5, 997.5, -0.5)
```



Analysis

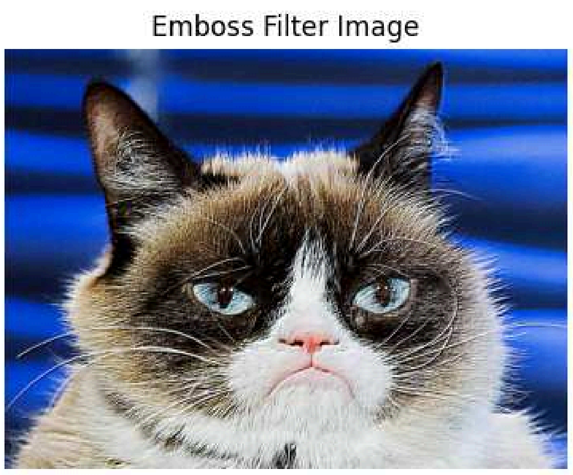
Using the blur filter in our original image, we can see that our output is got really blurred than the original. As shown on the output, the edges of our image is less visible and make it more smooth.

Our sharpening, edge detection, and blur filters use kernels that are highly symmetric. Sometimes, though, kernels with less symmetry produce an interesting effect. Let's consider a kernel that blurs on one side (with positive weights) and sharpens on the other (with negative weights). It will produce a ridged or embossed effect.

In [53]:
```
class EmbossFilter(VConvolutionFilter):
  """An emboss filter with a 1-pixel radius."""

  def __init__(self):
    kernel = np.array([[-2, -1, 0],
                       [-1, 1, 1],
                       [0, 1, 2]])
    VConvolutionFilter.__init__(self, kernel)
```

In [56]:
```
emboss_filter = EmbossFilter()
dst_emboss_color = np.zeros_like(src)
emboss_filter.apply(src,dst_emboss_color)

# Display the original and emboss filter images
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(cv2.cvtColor(src, cv2.COLOR_BGR2RGB))
axes[0].set_title("Original Colored Image")
axes[0].axis("off")

axes[1].imshow(cv2.cvtColor(dst_emboss_color, cv2.COLOR_BGR2RGB))
axes[1].set_title("Emboss Filter Image")
axes[1].axis("off")
```

Out[56]:
```
(-0.5, 1329.5, 997.5, -0.5)
```



Run this class and demonstrate the output. Provide an analysis

Analysis

In using the Emboss Filter in the `src` image we demonstrate how an emboss work. We can observed that the embossed image slightly enhance the edges as seen from its light and dark ares. This effect create a depth in the picture and loss its softness from the original image.

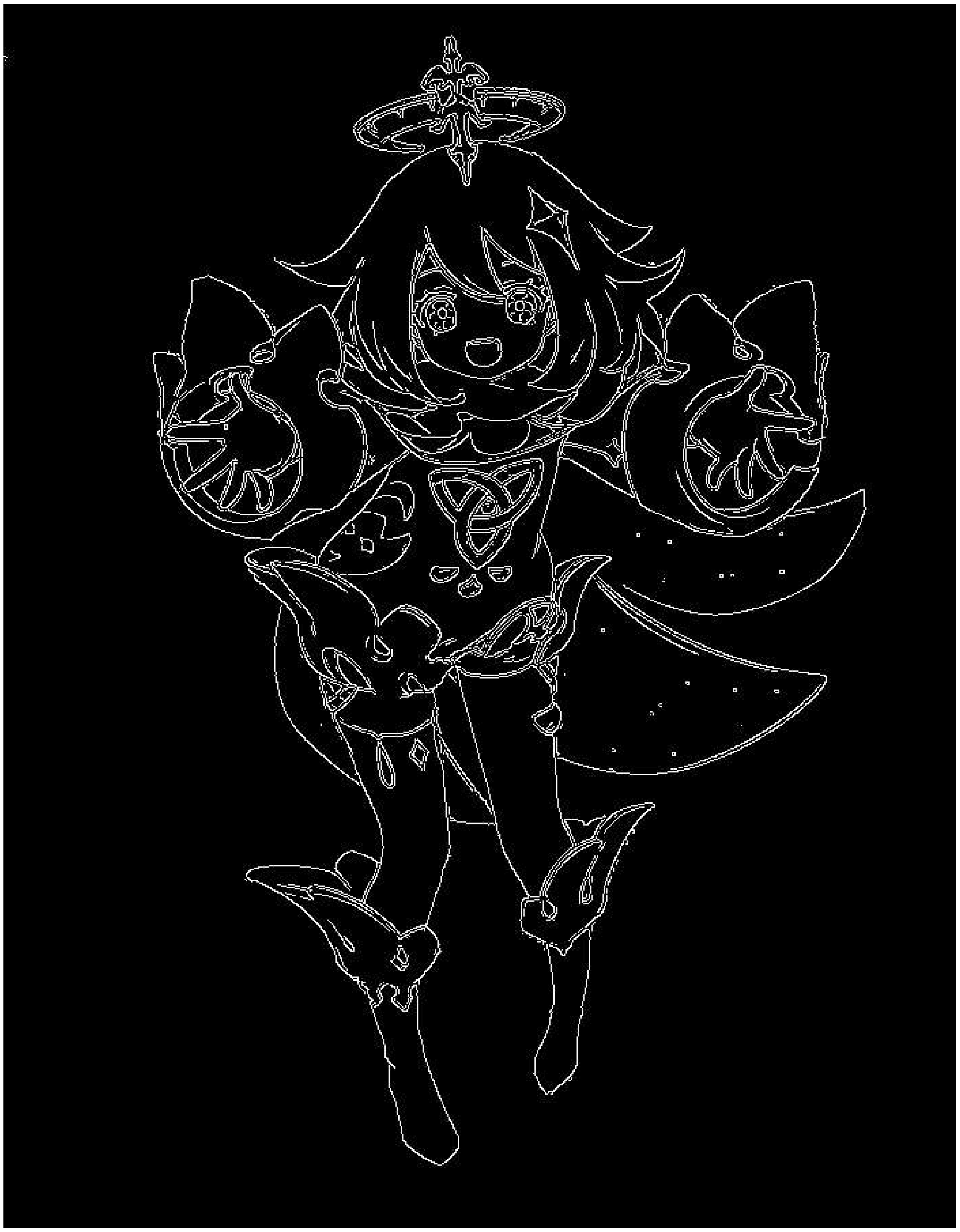
Edge Detection with Canny

OpenCV also offers a very handy function called Canny (after the algorithm's inventor, John F. Canny), which is very popular not only because of its effectiveness, but also the simplicity of its implementation in an OpenCV program, as it is a one-liner:

In [62]:
```
# import cv2
# import numpy as np
# from google.colab.patches import cv2_imshow

img = cv2.imread('/content/drive/MyDrive/CVdata/paimon.png', 0)
cv2.imwrite("canny.jpg", cv2.Canny(img, 200, 300))
```

Out[62]:
```
True
```

In [63]:
```
img_canny = cv2.imread('/content/canny.jpg')
cv2_imshow(img_canny)
```



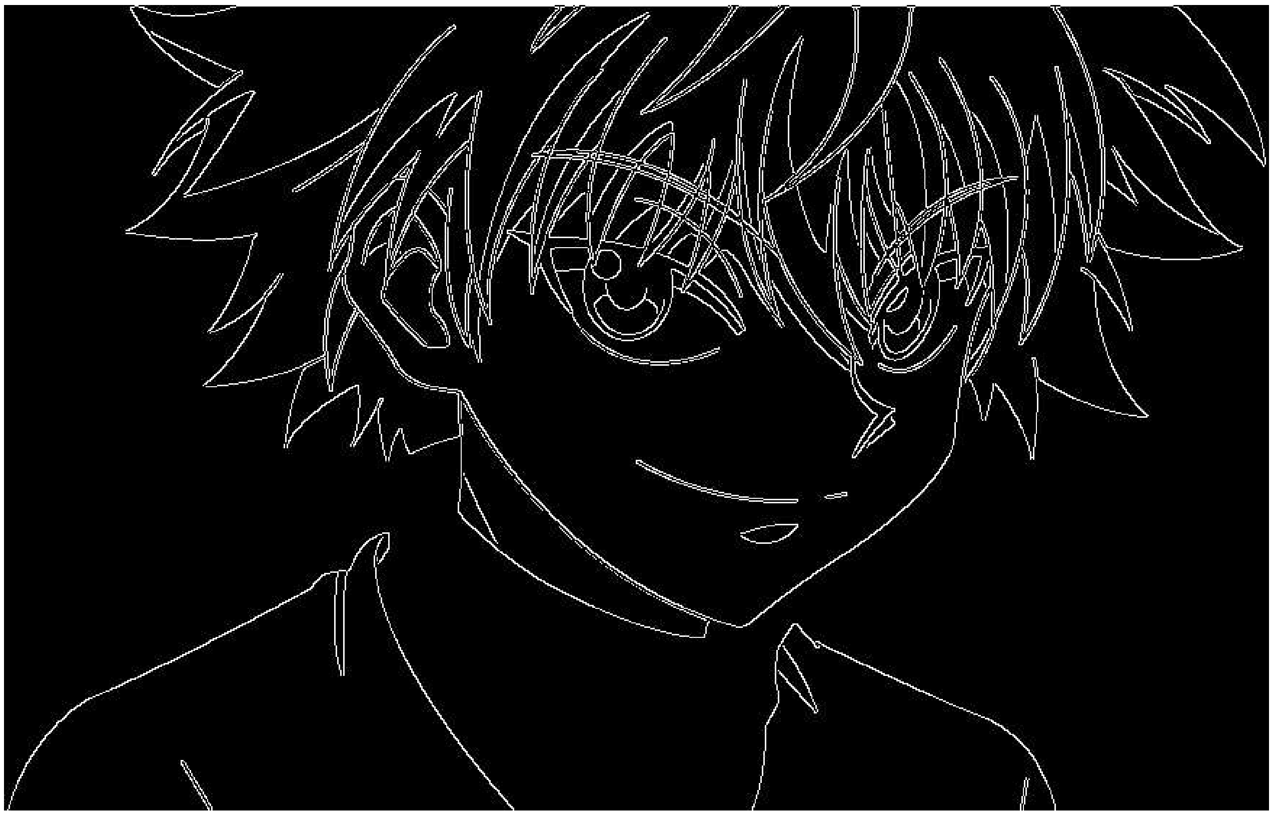
The Canny edge detection algorithm is quite complex but also interesting: it's a five-step process that denoises the image with a Gaussian filter, calculates gradients, applies non maximum suppression (NMS) on edges, a double threshold on all the detected edges to eliminate false positives, and, lastly, analyzes all the edges and their connection to each other to keep the real edges and discard the weak ones.

Try it on your own image, do you agree that it's an effective edge detection algorithm? Demonstrate your samples before making a conclusion

```
In [65]: # Using Canny Edge to simple Images
         killua = cv2.imread('/content/drive/MyDrive/CVdata/killua.png', 0)
         cv2.imwrite("SimpleCanny.jpg",cv2.Canny(killua, 200, 300))
```

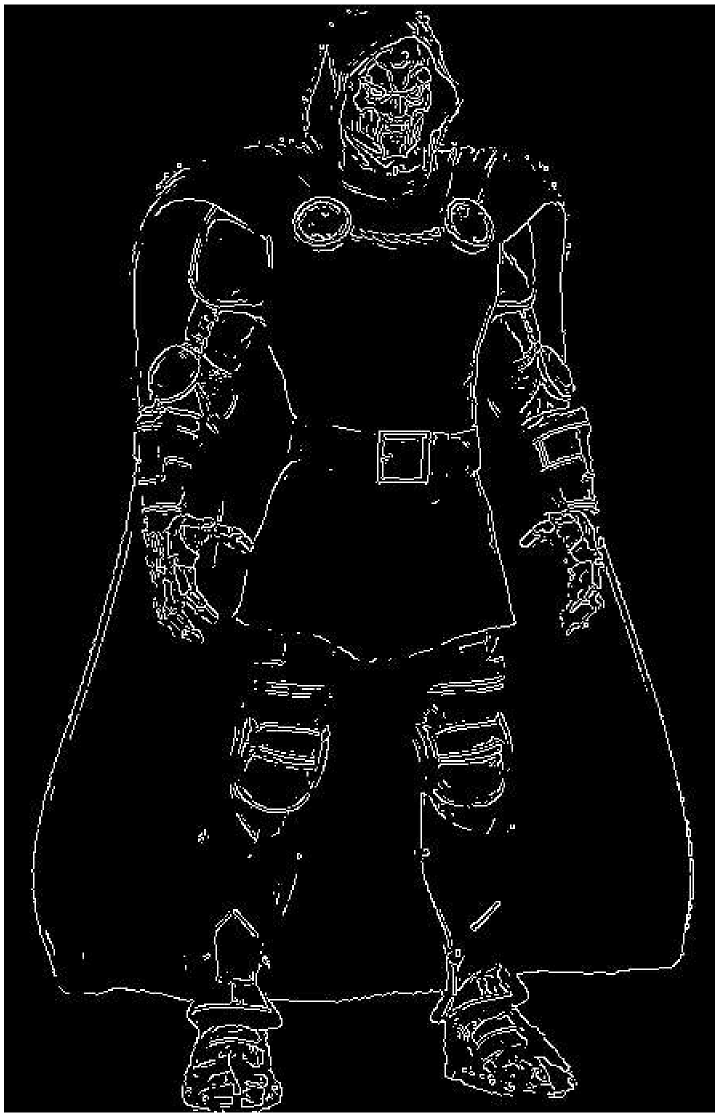
```
Out[65]: True
```

```
In [66]: img_killua = cv2.imread('/content/SimpleCanny.jpg')
         cv2_imshow(img_killua)
```



```
In [93]: #using Complex images
         doom = cv2.imread('/content/drive/MyDrive/CVdata/drdoom.png', 0)
         cv2.imwrite("ComplexCanny.jpg",cv2.Canny(doom, 200, 220))

Out[93]: True
```

```
In [94]: img_doom = cv2.imread('/content/ComplexCanny.jpg')
         cv2_imshow(img_doom)
```



Analysis

Canny edge Detection effectivley outlined some simple strong edges of the images. This indicate that the algorithm is suitable for sharp contrast and structure edges. Meanwhile in a more complex images with some weak and strong edges, the image output loss its some fine details due to its treshold as indicated in more complex image. Adjusting this parameters would likely to refine the results.

Contour Detection

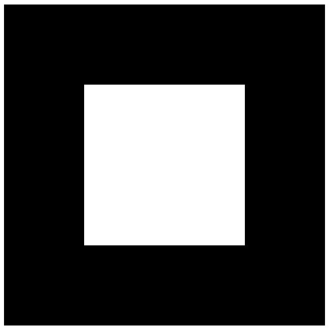
Another vital task in computer vision is contour detection, not only because of the obvious aspect of detecting contours of subjects contained in an image or video frame, but because of the derivative operations connected with identifying contours.

These operations are, namely, computing bounding polygons, approximating shapes, and generally calculating regions of interest, which considerably simplify interaction with image data because a rectangular region with NumPy is easily defined with an array slice. We will be using this technique a lot when exploring the concept of object detection (including faces) and object tracking.

In [95]:
```
# import numpy as np
# import cv2
# from google.colab.patches import cv2_imshow
```

```
img = np.zeros((200,200), dtype=np.uint8)
img[50:150, 50:150] = 255
```

In [96]:
```
cv2_imshow(img)
```

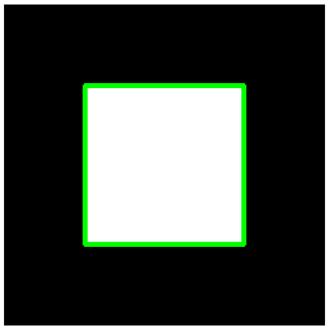


In [97]:
```
ret, thresh = cv2.threshold(img, 127, 255, 0)
print(ret)
print(thresh)
```

```
127.0
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

In [98]:
```
contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
color = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
```

In [99]:
```
img = cv2.drawContours(color, contours, -1, (0, 255, 0), 2)
cv2_imshow(color)
```



What is indicated by the green box in the middle? Why are there no other green lines? Provided an analysis

The green box in the middle indicate the boundary of iamge or ROI (region of interest). This green box identitfy the boundary or outline of our image. There are no other green lines because we select certain contours like hierarchy which only one image we have and it met the criteria. In the exercise, we only have on primarily object to be detect and it was highlighted.

Contours - bounding box, minimum area rectangle, and minimum enclosing circle

Finding the contours of a square is a simple task; irregular, skewed, and rotated shapes bring the best out of the cv2.findContours utility function of OpenCV. Consider this sample image:

In a real-life application, we would be most interested in determining the bounding box of the subject, its minimum enclosing rectangle, and its circle. The cv2.findContours function in conjunction with a few other OpenCV utilities makes this very easy to accomplish:

In [199...
```
# import cv2
# import numpy as np

img = cv2.pyrDown(cv2.imread("/content/drive/MyDrive/CVdata/hammer.png", cv2.IMREAD_UNCHANGED))

ret, thresh = cv2.threshold(cv2.cvtColor(img.copy(), cv2.COLOR_BGR2GRAY), 127, 255, cv2.THRESH_BINARY)
```

In [200...
```
cv2_imshow(img)
```

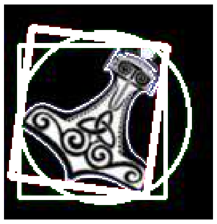
In [201...
```
contours, hier = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

In [202...
```
for c in contours:
  # Find the bounding box coordinates
  x, y, w, h = cv2.boundingRect(c)
  cv2.rectangle(img, (x,y), (x+w, y+h), (0, 255, 0), 2)

  # Find minimum area
  rect = cv2.minAreaRect(c)

  # Calculate coordinates of the minimum area rectangle
  box = cv2.boxPoints(rect)

  # Normalize coordinates to integers
  box = np.int0(box)

  # Draw contours
  cv2.drawContours(img, [box], 0, (0, 0, 255), 3)

  # Calculate center and radius of minimu enclosing circle
  (x, y), radius = cv2.minEnclosingCircle(c)

  # Cast to integers
  center = (int(x), int(y))
  radius = int(radius)

  # Draw the circle
  img = cv2.circle(img, center, radius, (0, 255, 0), 2)
```

```
<ipython-input-202-935b8ec75673>:13: DeprecationWarning: `np.int0` is a deprecated alias for `np.intp`.  (Dep
recated NumPy 1.24)
  box = np.int0(box)
```

In [203…

```
cv2.drawContours(img, contours, -1, (255, 0, 0), 1)
cv2_imshow(img)
```



Explain what has happened so far.

In [204…

```
x, y, w, h = cv2.boundingRect(c)
```

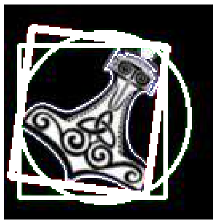
This is a pretty straightforward conversion of contour information to the (x, y) coordinates, plus the height and width of the rectangle. Drawing this rectangle is an easy task and can be done using this code:

In [205…

```
cv2.rectangle(img, (x,y), (x+w, y+h), (0, 255, 0), 2)
```

```
Out[205… array([[[  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 ...,
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255]],

                [[  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 ...,
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255]],

                [[  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 ...,
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255]],

                ...,

                [[  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 ...,
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255]],

                [[  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 ...,
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255]],

                [[  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 ...,
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255],
                 [  0,   0,   0, 255]]], dtype=uint8)
```

```
In [206… cv2_imshow(img)
```



Secondly, let's calculate the minimum area enclosing the subject:

```
In [207… rect = cv2.minAreaRect(c)
         box = cv2.boxPoints(rect)
         box = np.int0(box)
```

```
<ipython-input-207-ceb1eab3a452>:3: DeprecationWarning: `np.int0` is a deprecated alias for `np.intp`.  (Depr
ecated NumPy 1.24)
  box = np.int0(box)
```

The mechanism used here is particularly interesting: OpenCV does not have a function to calculate the coordinates of the minimum rectangle vertexes directly from the contour information. Instead, we calculate the minimum rectangle area, and then calculate the vertexes of this rectangle.

Note that the calculated vertexes are floats, but pixels are accessed with integers (you can't access a "portion" of a pixel), so we need to operate this conversion. Next, we draw the box, which gives us the perfect opportunity to introduce the cv2.drawContours function:

In [208…
```
cv2.drawContours(img, [box], 0, (0, 0, 255), 3)
```

Out[208…
```
array([[[  0,   0,   0, 255],
        [  0,   0,   0, 255],
        [  0,   0,   0, 255],
        ...,
        [  0,   0,   0, 255],
        [  0,   0,   0, 255],
        [  0,   0,   0, 255]],

       [[  0,   0,   0, 255],
        [  0,   0,   0, 255],
        [  0,   0,   0, 255],
        ...,
        [  0,   0,   0, 255],
        [  0,   0,   0, 255],
        [  0,   0,   0, 255]],

       [[  0,   0,   0, 255],
        [  0,   0,   0, 255],
        [  0,   0,   0, 255],
        ...,
        [  0,   0,   0, 255],
        [  0,   0,   0, 255],
        [  0,   0,   0, 255]],

       ...,

       [[  0,   0,   0, 255],
        [  0,   0,   0, 255],
        [  0,   0,   0, 255],
        ...,
        [  0,   0,   0, 255],
        [  0,   0,   0, 255],
        [  0,   0,   0, 255]],

       [[  0,   0,   0, 255],
        [  0,   0,   0, 255],
        [  0,   0,   0, 255],
        ...,
        [  0,   0,   0, 255],
        [  0,   0,   0, 255],
        [  0,   0,   0, 255]],

       [[  0,   0,   0, 255],
        [  0,   0,   0, 255],
        [  0,   0,   0, 255],
        ...,
        [  0,   0,   0, 255],
        [  0,   0,   0, 255],
        [  0,   0,   0, 255]]], dtype=uint8)
```

In [209…
```
cv2_imshow(img)
```

Firstly, this function—like all drawing functions—modifies the original image. Secondly, it takes an array of contours in its second parameter, so you can draw a number of contours in a single operation. Therefore, if you have a single set of points representing a contour polygon, you need to wrap these points into an array, exactly like we did with our box in the preceding example. The third parameter of this function specifies the index of the contours array that we want to draw: a value of -1 will draw all contours; otherwise, a contour at the specified index in the contours array (the second parameter) will be drawn.

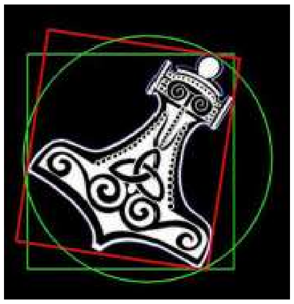
Most drawing functions take the color of the drawing and its thickness as the last two parameters.

The last bounding contour we're going to examine is the minimum enclosing circle:

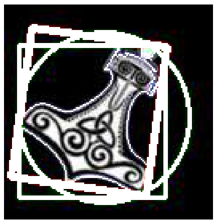
In [210…
```
(x, y), radius = cv2.minEnclosingCircle(c)
center = (int(x), int(y))
radius = int(radius)
img = cv2.circle(img, center, radius, (0, 255, 0), 2)
```

The only peculiarity of the cv2.minEnclosingCircle function is that it returns a two-element tuple, of which the first element is a tuple itself, representing the coordinates of the circle's center, and the second element is the radius of this circle. After converting all these values to integers, drawing the circle is quite a trivial operation.

Show the final image output and provide an analysis. Does it look similar to the image shown below?

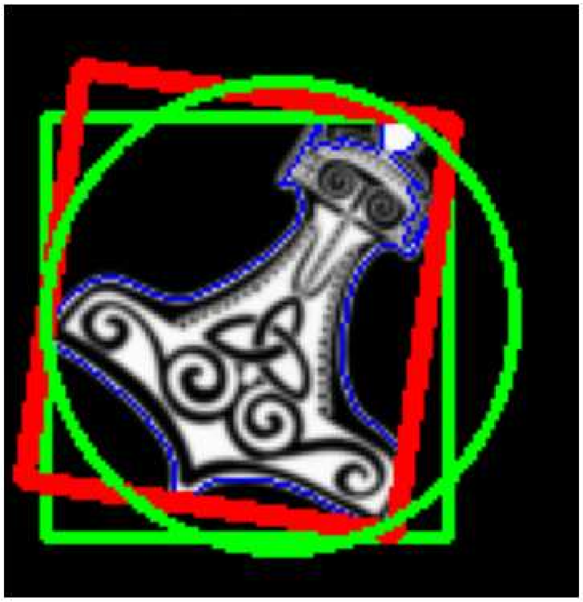


In [213…
```
cv2_imshow(img)
```



In [212…
```
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.axis("off")
```

Out[212…
```
(-0.5, 129.5, 133.5, -0.5)
```



Analysis

The output we got in the activity is similar to the expected output shown above. The code provide a simple shaped contours for the hammer image.

4. Supplementary Activity

For this section of the activity, you must have your favorite fictional character's picture ready.

Perform/Answer the following:

- Run all classes above meant to filter an image to your favorite character.
- Use edge detection and contour detection on your fave character. Do they indicate the same?
- Modify your character's picture such that bounding boxes similar to what happens in the last procedure will be written on the image.
- Research on the benefits of using canny and contour detection. What happens to the image after edge detection? What happens when you apply contour straight after?

Filter Application to Dr. Doom

In [214...
```
src = cv2.imread('/content/drive/MyDrive/CVdata/drdoom.png')
dst = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)
```

In [215...
```
cv2_imshow(src)
```

In [216…

```
cv2_imshow(dst)
```

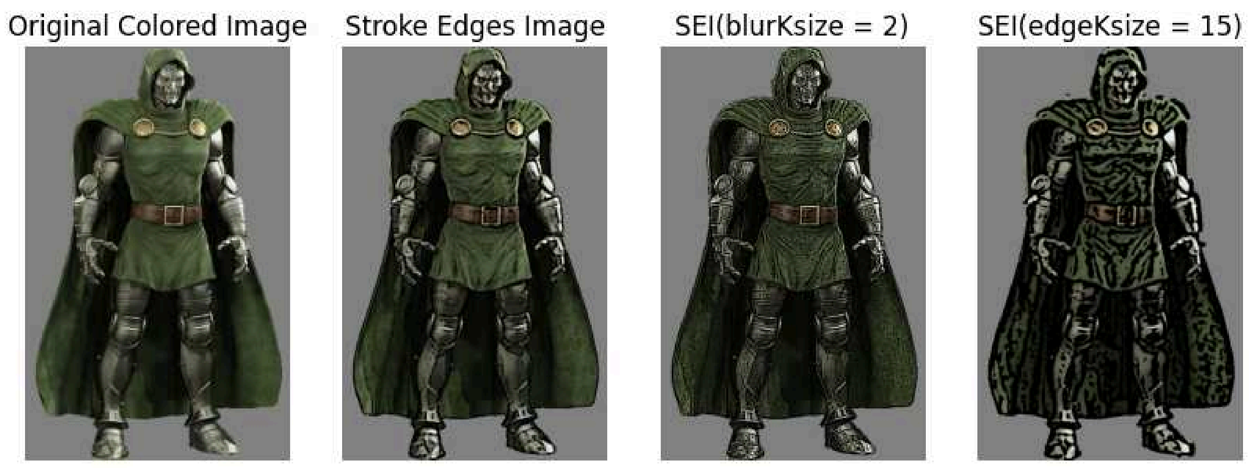
Stroke Edges Sample Output

In [228…

```python
new_doom_img = strokeEdges(src,dst)
new_blur_doom_img = strokeEdges(src,dst,blurKsize= 2)
new_edge_doom_img = strokeEdges(src,dst,edgeKsize= 15)


fig, axes = plt.subplots(1, 4, figsize=(10, 5))
axes[0].imshow(cv2.cvtColor(src, cv2.COLOR_BGR2RGB))
axes[0].set_title("Original Colored Image")
axes[0].axis("off")

axes[1].imshow(cv2.cvtColor(new_doom_img, cv2.COLOR_BGR2RGB))
axes[1].set_title("Stroke Edges Image")
axes[1].axis("off")

axes[2].imshow(cv2.cvtColor(new_blur_doom_img, cv2.COLOR_BGR2RGB))
axes[2].set_title("SEI(blurKsize = 2)")
axes[2].axis("off")

axes[3].imshow(cv2.cvtColor(new_edge_doom_img, cv2.COLOR_BGR2RGB))
axes[3].set_title("SEI(edgeKsize = 15)")
axes[3].axis("off")
```

Out[228…

```
(-0.5, 443.5, 691.5, -0.5)
```



filter2d() Sample Output

In [231…
```
new_doom_img2 = cv2.filter2D(src, -1, kernel, dst)
```

In [232…
```
cv2_imshow(new_doom_img2)
```

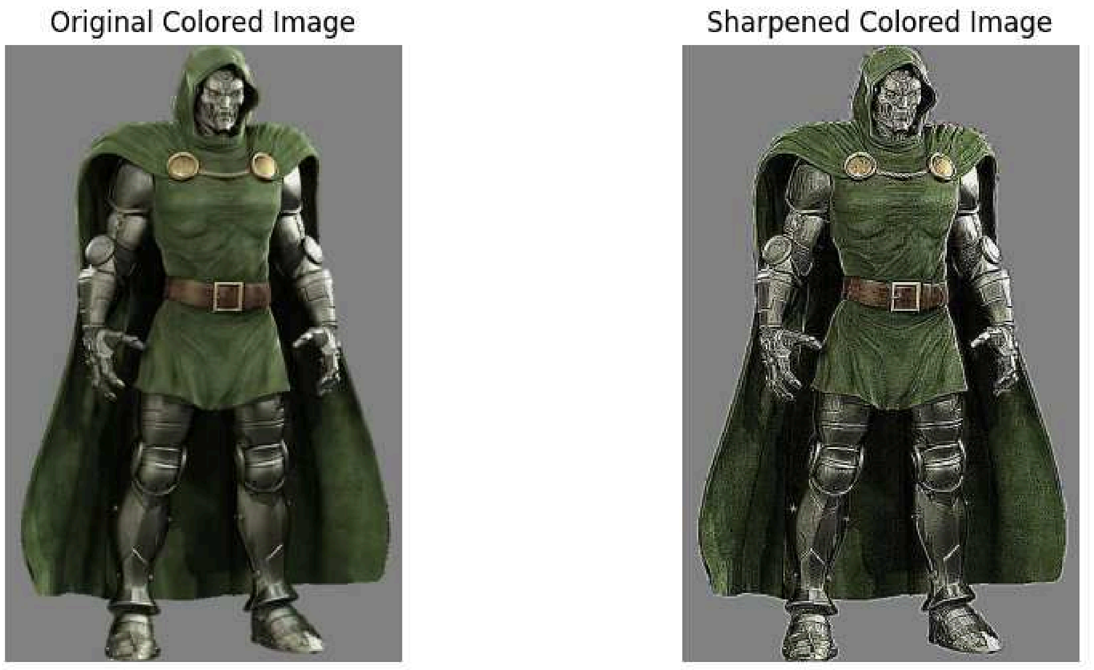
Shapen Filter Output

In [233…

```
sharpen_filter = SharpenFilter()

sharpened_img = src.copy()
sharpen_filter.apply(src,sharpened_img)

# Display the original and sharpened colored images
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(cv2.cvtColor(src, cv2.COLOR_BGR2RGB))
axes[0].set_title("Original Colored Image")
axes[0].axis("off")

axes[1].imshow(cv2.cvtColor(sharpened_img, cv2.COLOR_BGR2RGB))
axes[1].set_title("Sharpened Colored Image")
axes[1].axis("off")
```

Out[233…

```
(-0.5, 443.5, 691.5, -0.5)
```



Find Edge Filter Output

In [236…

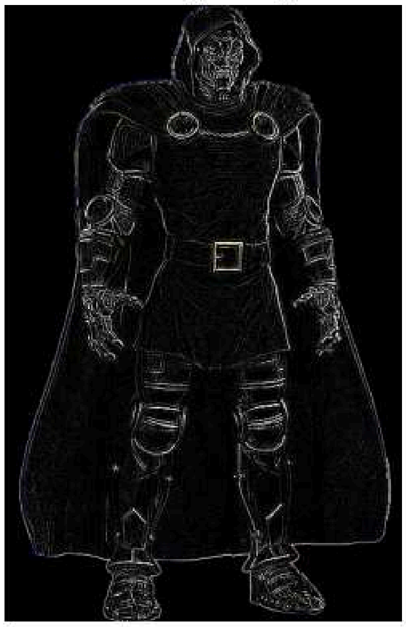
```
findedges_filter = FindEdgesFilter()

findedges_img = src.copy()
findedges_filter.apply(src,findedges_img)

# Display the original and edge filter  images
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(cv2.cvtColor(src, cv2.COLOR_BGR2RGB))
axes[0].set_title("Original Colored Image")
axes[0].axis("off")

axes[1].imshow(cv2.cvtColor(findedges_img, cv2.COLOR_BGR2RGB))
axes[1].set_title("Find Edge Image")
axes[1].axis("off")
```

Out[236…

```
(-0.5, 443.5, 691.5, -0.5)
```

Original Colored Image

Find Edge Image

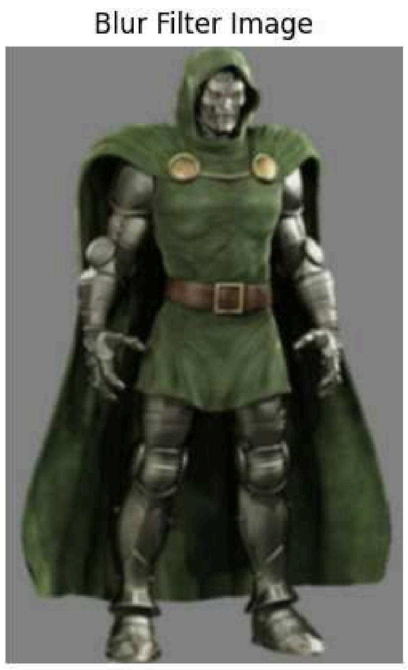


Blur Filter Ouput

In [237…
```
blur_filter = Blurfilter()

blur_img = src.copy()
blur_filter.apply(src,blur_img)

# Display the original and blur filter images
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(cv2.cvtColor(src, cv2.COLOR_BGR2RGB))
axes[0].set_title("Original  Image")
axes[0].axis("off")

axes[1].imshow(cv2.cvtColor(blur_img, cv2.COLOR_BGR2RGB))
axes[1].set_title("Blur Filter Image")
axes[1].axis("off")
```

Out[237…
```
(-0.5, 443.5, 691.5, -0.5)
```

Original Image



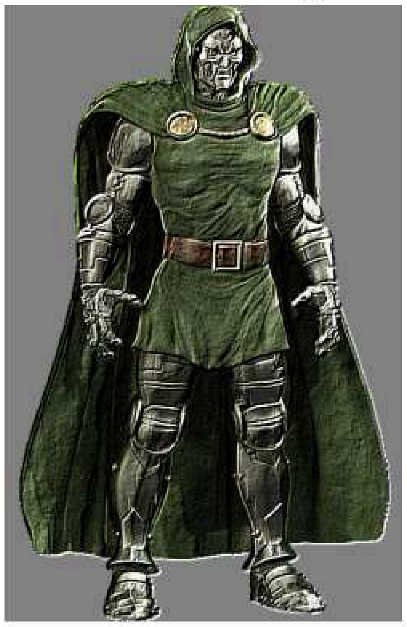
Emboss Filter Image

```
In [238…
emboss_filter = EmbossFilter()

emboss_img = src.copy()
emboss_filter.apply(src,emboss_img)

# Display the original and emboss filter images
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(cv2.cvtColor(src, cv2.COLOR_BGR2RGB))
axes[0].set_title("Original Colored Image")
axes[0].axis("off")

axes[1].imshow(cv2.cvtColor(emboss_img, cv2.COLOR_BGR2RGB))
axes[1].set_title("Emboss Filter Image")
axes[1].axis("off")
```

```
Out[238…  (-0.5, 443.5, 691.5, -0.5)
```

Original Colored Image

Emboss Filter Image



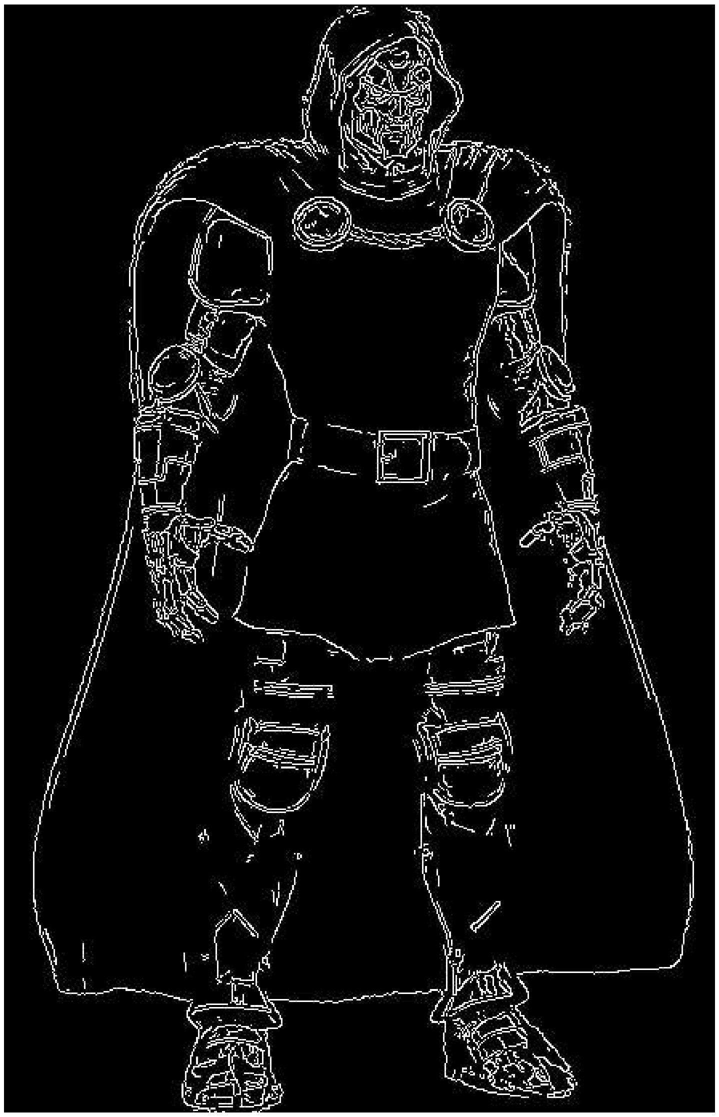
Contour and Edge Application to Dr.Doom

Edge

```
In [270…
doom_img = cv2.imread('/content/drive/MyDrive/CVdata/drdoom.png')
cv2.imwrite('DrDoomCanny.jpg', cv2.Canny(doom_img, 150, 200))
```

```
Out[270…
True
```

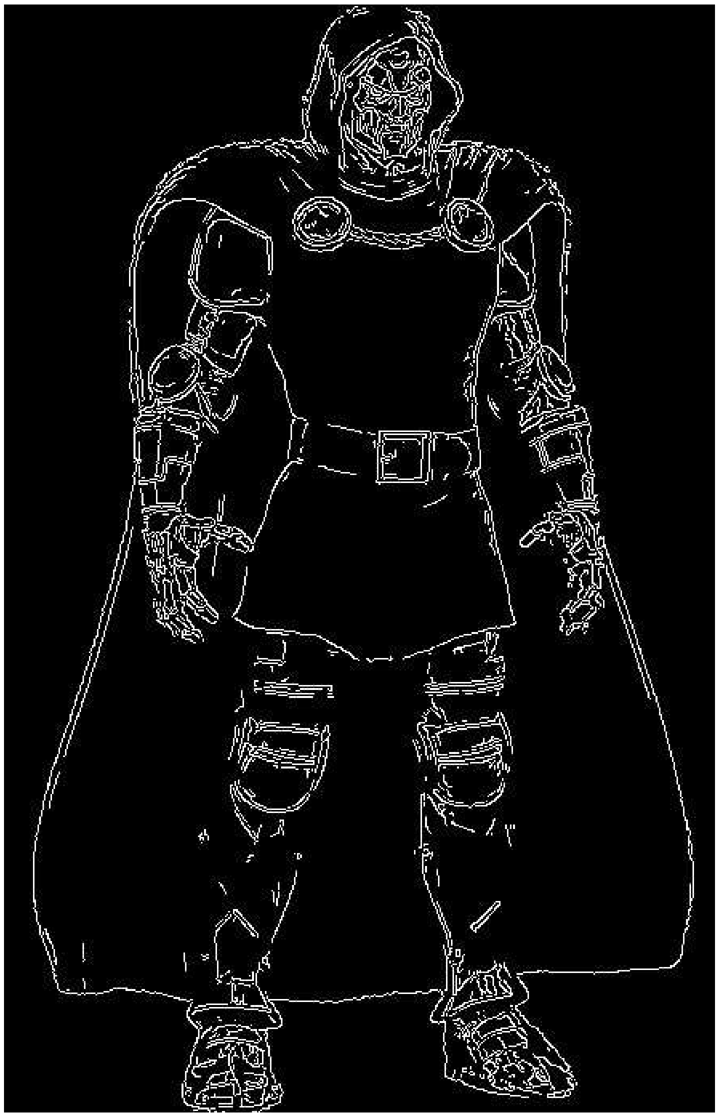
```
In [271…
img_doom = cv2.imread('/content/DrDoomCanny.jpg')
cv2_imshow(img_doom)
```



Contour

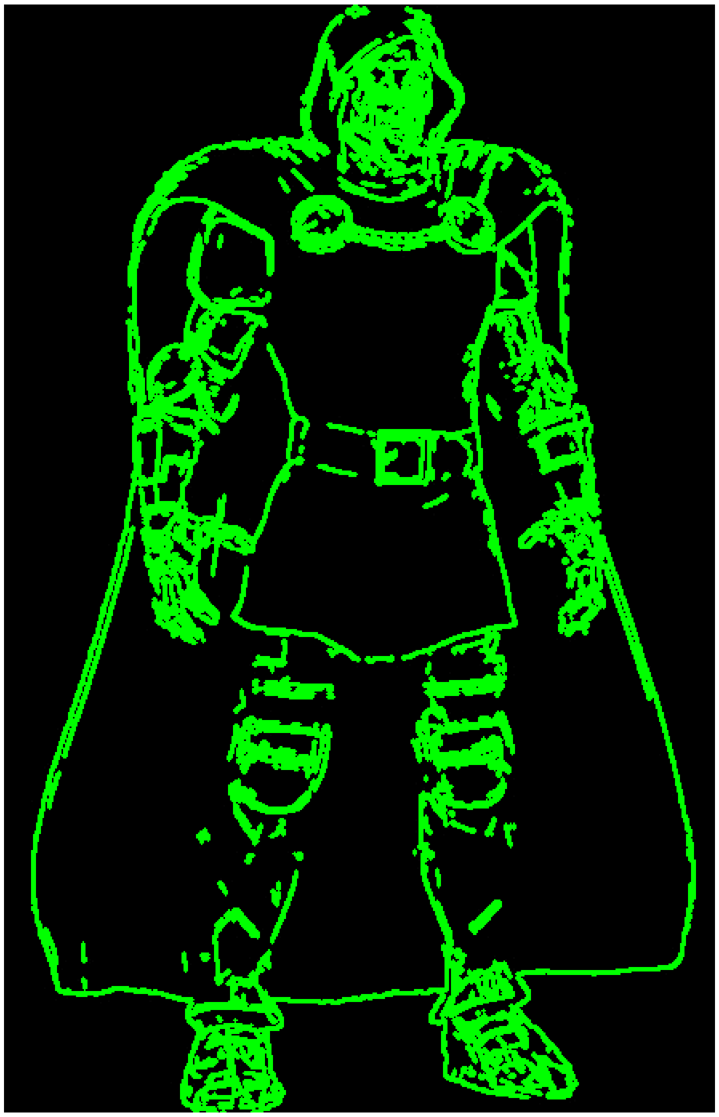
In [282…

```
img = cv2.imread('/content/DrDoomCanny.jpg',cv2.IMREAD_GRAYSCALE)
cv2_imshow(img)

# Apply thresholding to the grayscale image 'img'
ret, thresh = cv2.threshold(img, 127, 255, 0)

# Now 'thresh' will be a single-channel image suitable for findContours
contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
color = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
```



In [283…

```python
img = cv2.drawContours(color, contours, -1, (0, 255, 0), 2)
cv2_imshow(color)
```

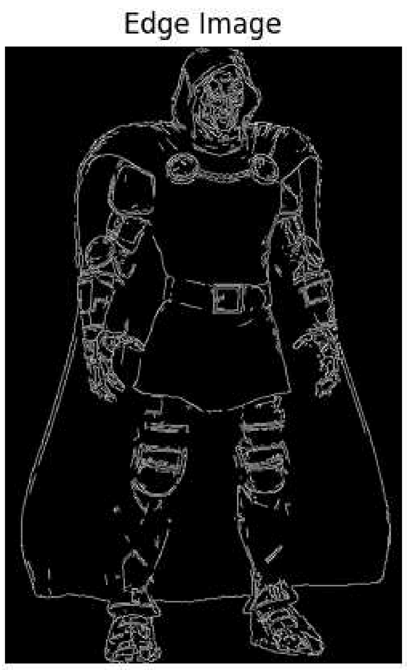


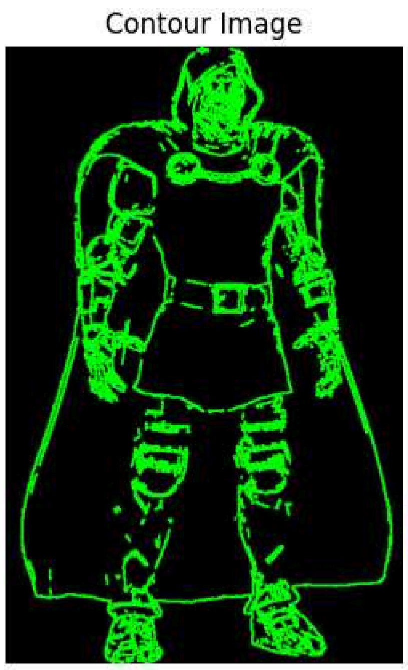
In [286…

```
# Display the Edge and Contour images
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(cv2.cvtColor(img_doom, cv2.COLOR_BGR2RGB))
axes[0].set_title("Edge Image")
axes[0].axis("off")

axes[1].imshow(cv2.cvtColor(color, cv2.COLOR_BGR2RGB))
axes[1].set_title("Contour Image")
axes[1].axis("off")
```

Out[286…

```
(-0.5, 443.5, 691.5, -0.5)
```





Contour with Bounding Box Application

In [290…
```
img = cv2.pyrDown(cv2.imread('/content/drive/MyDrive/CVdata/drdoom.png'),cv2.IMREAD_GRAYSCALE)

ret, thresh = cv2.threshold(cv2.cvtColor(img.copy(), cv2.COLOR_BGR2GRAY), 127, 255, cv2.THRESH_BINARY)
```

In [291…
```
cv2_imshow(img)
```

In [292…
```
contours, hier = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

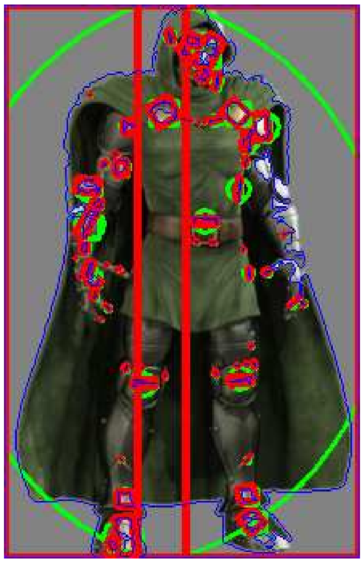
In [295…
```
for c in contours:
  # Find the bounding box coordinates
  x, y, w, h = cv2.boundingRect(c)
  cv2.rectangle(img, (x,y), (x+w, y+h), (0, 255, 0), 2)

  # Find minimum area
  rect = cv2.minAreaRect(c)

  # Calculate coordinates of the minimum area rectangle
  box = cv2.boxPoints(rect)
```

```
    # Normalize coordinates to integers
    box = np.int0(box)

    # Draw contours
    cv2.drawContours(img, [box], 0, (0, 0, 255), 3)

    # Calculate center and radius of minimu enclosing circle
    (x, y), radius = cv2.minEnclosingCircle(c)

    # Cast to integers
    center = (int(x), int(y))
    radius = int(radius)

    # Draw the circle
    img = cv2.circle(img, center, radius, (0, 255, 0), 2)
```

```
<ipython-input-295-7125e5b9682f>:13: DeprecationWarning: `np.int0` is a deprecated alias for `np.intp`.  (Dep
recated NumPy 1.24)
  box = np.int0(box)
```

In [296…

```
cv2.drawContours(img, contours, -1, (255, 0, 0), 1)
cv2_imshow(img)
```



Benefits of using canny and contour detection.

Canny Edge detection is a technique on image processing which detect the edges in images. It recognize both strong and weak lines of the image which is good on outlining shapes and details. It also reduce variation of brightess and color, preventing false edges during the process. Meanwhile, Contour detection is use to identify or recognize object. It is useful for maching shapes within an image which are essentially usefu for object detection.

After edge detection, the image transformed into binary where edges are highlighted. During this process, it will help use indentify the boundaries of an object and edges represent thin lines that outline the shape of an image.

Applying contour detection straight after edge detection will identify and outlined the contours of our object. It will trace the edge detected and form a continuous line which represent the boundary of our ROI resulting to more detailed object representation within an image.

5. Summary, Conclusions and Lessons Learned

In this notebook we perform various task on edge and contour detection as well as image processing manipulation such as filter, blur and edge. We also mange to use the apply what we learn in the exercises in our supplementary activity. I also compare the original image from the filter we used. The challenging part of this activity is changing the parameters of contours and edges to detect the object since my image is complex which is hard for the program to identify its edge lines.
