Le traitement d'images avec OpenCV

OpenCV (pour Open Computer Vision) est une bibliothèque graphique libre, initialement développée par Intel, spécialisée dans le traitement d'images en temps réel. La société de robotique Willow Garage et la société ItSeez se sont succédé au support de cette bibliothèque. Depuis 2016 et le rachat de ItSeez par Intel, le support est de nouveau assuré par Intel. Cette bibliothèque est distribuée sous licence BSD et peut être trouvé sur GitHub (https://github.com/opencv/opencv)

OpenCV propose la plupart des opérations classiques en traitement bas niveau des images:

- lecture, écriture et affichage d'une image
- calcul de l'histogramme des niveaux de gris ou d'histogrammes couleurs
- lissage, filtrage
- seuillage d'image (méthode d'Otsu, seuillage adaptatif)

Sources et références:

- OpenCV (https://fr.wikipedia.org/wiki/OpenCV) sur Wikipédia
- Practical Python and OpenCV (https://www.pyimagesearch.com/practical-python-opencv/) par Adrian Rosebrock
- OpenCV.org (https://opencv.org), le site officiel

Utiliser OpenCV sur le Raspberry Pi

Il faut se placer dans l'environnement virtuel à l'aide du terminal:
```
$source start_py3cv3.sh
```

Puis se placer dans le dossier où l'on veut créer notre notebook avec:
```
$cd
```

Ensuite, il faut importer les modules nécessaires OpenCV (cv2), pour utiliser le calcul matriciel (numpy) et pour l'affichage des images (matplotlib) dans le notebook Jupyter.

In [1]:

```
%matplotlib inline
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

Pour faire apparraître les images dans un notebook Jupyter définit la fonction suivante:

In [3]:

```
def show(img):
    #vérification de la dimension de l'image (dim3=couleur; dim2=noir et blanc)
    if len(img.shape)==3:
        img2 = cv2.cvtColor(img, cv2.COLOR_BGR2RGB) #convertion du mode de couleur
        plot = plt.imshow(img2)
    else:
        plot = plt.imshow(img, cmap = 'gray')
```

Pour faire apparaître les histograms on définit la fonction suivante:

In [4]:

```
def histogram(hist, title):
    plt.figure()
    plt.title(title)
    plt.xlabel("Bins") #Nom échelle des x
    plt.ylabel("# of Pixels") #Nom échelle des y
    plt.plot(hist)
    plt.xlim([0, 256])
    plt.show()
```

Système de couleur BVR

Les couleurs sont représenté par leur composantes bleu (B), vert (V) et rouge (R). Par rapport au standard RVB le plus répandu, dans OpenCV, les couleurs R et B sont inversées. Dans OpenCV les couleurs sont dans l'ordre BVR. Par exemple les couleurs de base sont:

In [7]:

```
NOIR = (0, 0, 0)
BLANC = (225, 225, 225)
ROUGE = (0, 0, 225)
VERT = (0, 225 ,0)
BLEU = (255, 0, 0)

ROUGE
```

Out[7]:

```
(0, 0, 225)
```

Lire et afficher une image

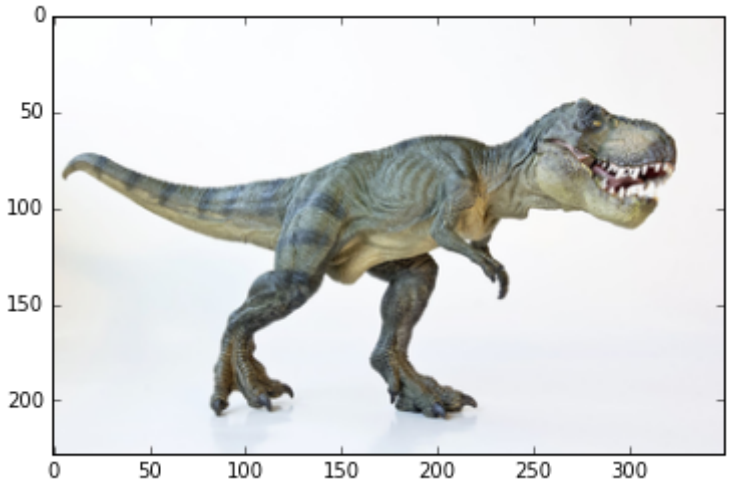
- img.shape: donne les dimensions de l'image (y, x, nombre de canals).
- La fonction `cv2.imread('abc.jpg')` lit l'image mais ne l'affiche pas.
- La fonction `cv2.imshow('abc.jpg')` l'image dans une nouvelle fenêtre
- Position des pixels:
  -(0,0) le pixel se trouve tout en haut à gauche de l'image.
  -(3,7) le pixel se trouve sur la troisième colonne en partant de la droite et sur la septième ligne en partant du haut.

-(x_{max},0) le pixel se trouve tout en haut à droite.
-(0,y_{max}) le pixel se trouve tout en bas à gauche.
(x_{max},y_{max}) le pixel se trouve tout en bas à droite.

In [5]:

```
img = cv2.imread('trex.png')
print(img.shape)
show(img)
```

```
(228, 350, 3)
```



Fonctions de dessins dans OpenCV

Source: https://docs.opencv.org/3.1.0/dc/da5/tutorial_py_drawing_functions.html (https://docs.opencv.org/3.1.0/dc/da5/tutorial_py_drawing_functions.html)

OpenCV dispose des fonctions graphiques suivantes:

- `cv2.line()` pour dessiner der lignes
- `cv2.circle()` pour dessiner des cercles
- `cv2.rectangle()` pour dessiner des rectangles
- `cv2.ellipse()` pour dessiner des ellipses
- `cv2.putText()` pour ajouter du texte

Dessiner une ligne

Il faut d'abord créer une image noir, dans laquelle nous allons ajouter des éléments graphiques tel que

In [9]:

```
img = np.zeros((150, 300, 3), np.uint8)
cv2.line(img, (10, 10), (100, 100), ROUGE, 10)
```

Out[9]:

```
array([[[254, 254, 254],
        [254, 254, 254],
        [254, 254, 254],
        ...,
        [238, 239, 243],
        [237, 238, 242],
        [226, 227, 234]],

       [[254, 254, 254],
        [254, 254, 254],
        [254, 254, 254],
        ...,
        [238, 239, 243],
        [237, 238, 242],
        [231, 232, 238]],

       [[254, 254, 254],
        [254, 254, 254],
        [254, 254, 254],
        ...,
        [238, 239, 243],
        [237, 238, 242],
        [234, 235, 240]],

       ...,
       [[246, 241, 238],
        [246, 241, 238],
        [246, 241, 238],
        ...,
        [242, 237, 236],
        [242, 237, 236],
        [242, 237, 236]],

       [[246, 241, 238],
        [246, 241, 238],
        [246, 241, 238],
        ...,
        [243, 238, 237],
        [243, 238, 237],
        [243, 238, 237]],

       [[247, 242, 239],
        [247, 242, 239],
        [247, 242, 239],
        ...,
        [243, 238, 237],
        [243, 238, 237],
        [243, 238, 237]]], dtype=uint8)
```

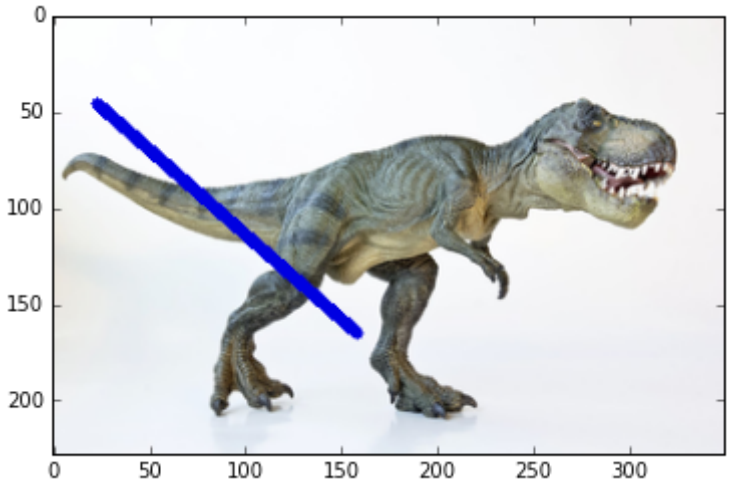
La fonction OpenCV pour dessiner une ligne a cette forme:
`cv2.line(img, (x1,y1), (x2, y2), (B, G, R), e)`

Les paramètres signifient:

- `img` = nom de l'image sur laquelle la ligne va être dessinée
- `(x1, y1)` = position du point (pixel) au début de la ligne
- `(x2, y2)` = position du point (pixel) à la fin de la ligne
- `(B, G, R)` = couleur de la ligne
- `e` = épaisseur de la ligne (facultatif)

In [56]:

```
# Exemple:
img1 = cv2.line(img1, (23,46),(158,165), (225,0,0), 5)
show(img1) # Ne pas oublier d'afficher l'image
```



Dessiner un rectangle

In [14]:

```
#Si l'on veut faire un dessin sur une autre image il faut redéfinir son image:
img2 = cv2.imread('trex.png')
```

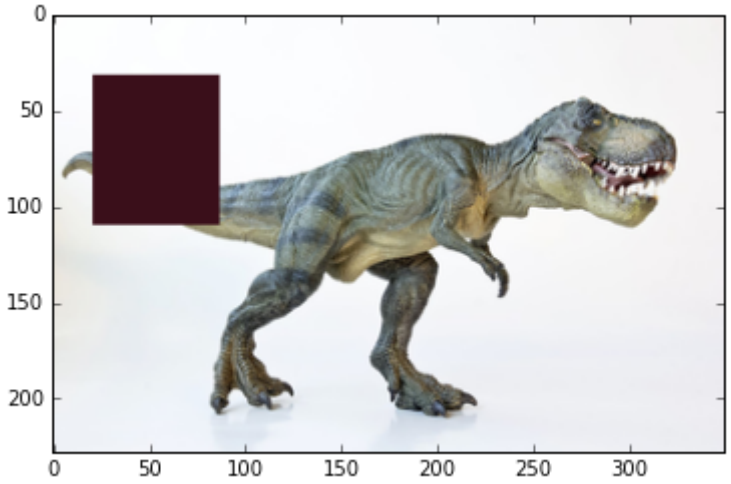
La fonction OpenCV pour dessiner un rectangle a cette forme:
`cv2.rectangle(img, (x1,y1), (x2, y2), (B, G, R), e)`

Les paramètres signifient:

- `img` = nom de l'image sur laquelle la ligne va être dessinée
- `(x1, y1)` = position du point (pixel) du coin en haut à gauche
- `(x2, y2)` = position du point (pixel) du coin en bas à droite
- `(B, G, R)` = couleur des bordures du rectangle
- `e` = épaisseur des bordures du rectangle (valeur négative = rectangle rempli)

In [15]:

```
# Exemple:
img2 = cv2.rectangle(img2, (21,32), (86,109),(26,15,58), -1)
show(img2)
```



Dessiner un cercle

In [16]:

```
img3 = cv2.imread("trex.png")
```

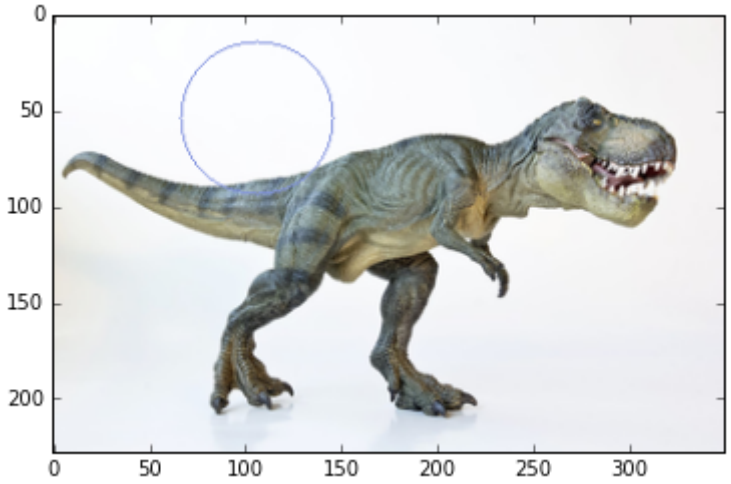
Ecrire la fonction pour faire apparaître un cercle:

cv2.circle(img, (x,y), r, (B,G,R), e)

img = nom de l'image sur laquelle le cercle doit apparaître
(x,y) = position du centre du cercle
r = rayon du crcle
(B,G,R) = couleur des bordures du cercle
e (facultatif) = épaisseur des bordures du cercle (valeur négative = cercle rempli)

In [17]:

```
# Exemple:
cv2.circle(img3, (106,54), 40,(223,145,128))
show(img3)
```



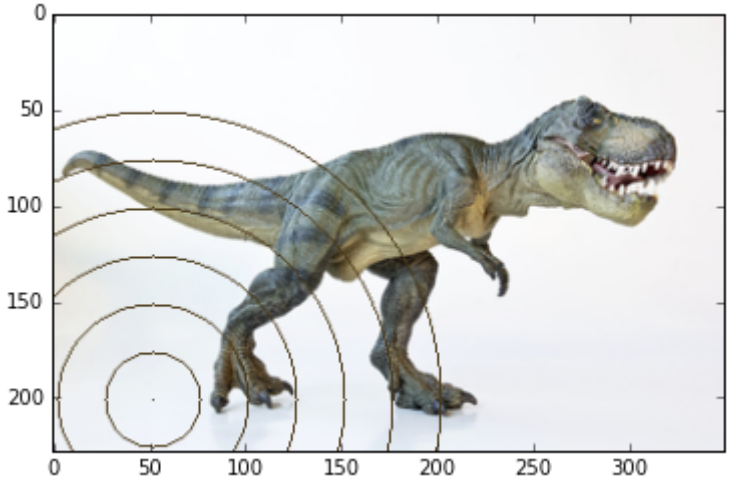
Si l'on veut dessiner plusieurs cercles, il faut écrire la fonction suivante:

In [18]:

```
img4 = cv2.imread("trex.png")

def plusieurs_cercles(x,y,r,B,G,R,e):
    for r in range (0, 175, 25): # 0 = rayon de départ; 175 = rayon final, 25 = pallier
        cv2.circle(img4, (x,y), r,(B,G,R), e)
        show(img4)

plusieurs_cercles(52,201,2,22,56,74,1)
```



Transformations géométrique

Translation

Définition: une translation consiste à déplacer l'image de t_x et t_y pixels.

Pour faire une translation, il faut créer une matrice de translation:

$$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{pmatrix}$$

Voici comment créer une tel matrice dans numpy:

In [16]:

```
tx = 40
ty = 60
M = np.float32([[1, 0, tx], [0, 1, ty]])
print(M)
```

```
[[ 1.  0. 40.]
 [ 0.  1. 60.]]
```

Ensuite il faut appeller la fonction qui réalise la translation à l'aide de la matrice:
`cv2.warpAffine(img, M, (img.shape[1], img.shape[0])`

Les paramètres

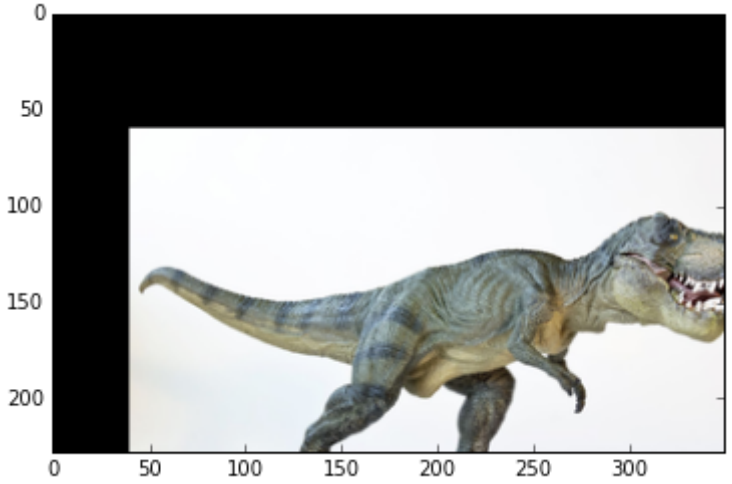
- `img` = image à translater
- `M` matrice de translation
- `img.shape[1], img.shape[0]` dimensions de l'image

In [33]:

```
# Exemple:
img5 = cv2.imread('trex.png')

M = np.float32([ [1,0,40] , [0,1,60] ])
img5 = cv2.warpAffine(img5, M, (img.shape[1], img.shape[0]))
show(img5)

# L'image est déplacée de 45 pixels vers la droite et de 60 pixels vers le bas
```



Rotation

Definition: la rotation consiste à tourner l'imagee de θ degrés autour d'un point.

Il faut donc définir un point de rotation:
point = (x,y)

Ensuite on calcule la matrice de rotation:
`M = cv2.getRotationMatrix2D(point, theta, scale)`

- `point` est le center de rotation
- `theta` est l'angle de rotation (en degrée)
- `scale` est l'echelle de l'image, ainsi avec 1.0 elle garde la même taille, mais avec 2.0 elle sera deux fois plus grande

La fonction `warpAffine` (application affine) qui effectue la rotation est la même que celle utilisée pour la translation. La différence se trouve donc dans la matrice.

In [12]:

```
center = (100, 100)
theta = 45
M = cv2.getRotationMatrix2D(center, 45, 1.0)
print(M)
```

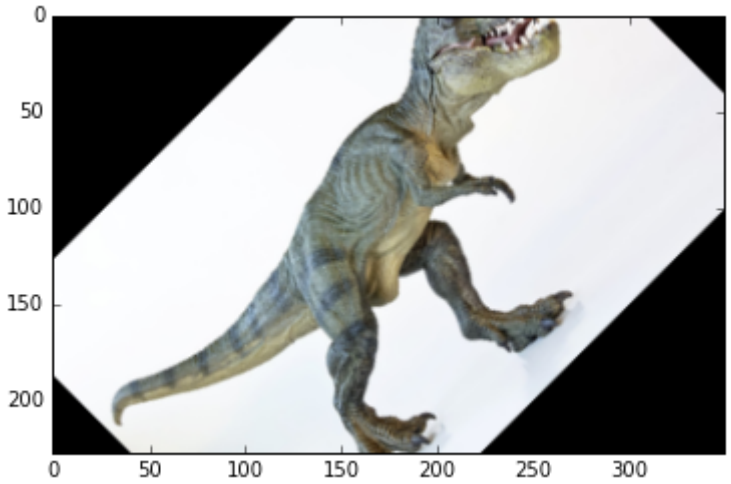
```
[[  0.70710678   0.70710678 -41.42135624]
 [ -0.70710678   0.70710678 100.        ]]
```

In [7]:

```
# Exemple:
img6 = cv2.imread('trex.png')

point = (img.shape[1]//2, img.shape[0]//2) # Milieu de l'image
M = cv2.getRotationMatrix2D(point, 45, 1.0)
img6 = cv2.warpAffine(img6, M, (img.shape[1], img.shape[0]))
show(img6)

# L'image est tournée de 45° à partir du centre. Les rotations suivent le sens du cercl
```



Redimensionnement

Définition: Le redimensionnement consiste à redéfinir les dimensions de l'image (augmenter la hauteur ou la largeur de l'image.

Il faut définir le ratio de changement:
ex: `r = n/img.shape[1]` (n est un float)

Ensuite on définit les nouvelles dimensions:
ex: `dim = (n, int(img.shape[0]*r))`

La fonction OpenCV pour redimensionner une image a la forme:

`cv2.resize(img, dim, interpolation = cv2.INTER_AREA)`

`interpolation = cv2.INTER_AREA` : est un algorithme de redimension

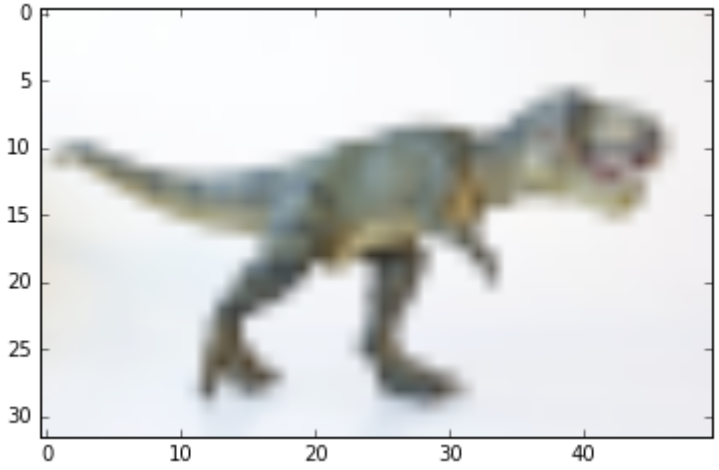
Dans cet exemple, l'image est redimensionner en fonction de la largeur, si l'on voulait la redimensionner en fonction de la hauteur il suffirait de permuter `img.shape[1]` et `img.shape[0]` de r et dim, sans modifier la fonction.

In [35]:

```
# Exemple:
img7 = cv2.imread('trex.png')

r = 50.0/img.shape[1]
dim = (50, int(img.shape[0]*r))
img7 = cv2.resize(img7, dim, interpolation = cv2.INTER_AREA)
show(img7)

#L'image a été redimensionner comme on peut voir les echelles des x et des y, elle est
```



Tourner une image

Définition: cette fonction consiste à retourner l'image selon un axe. Par exemple si l'on flip l'image du trex selon l'axe y, la tête du trex se retrouvera à la place de sa queue et inversement.

La fonction OpenCV qui tourne l'image a la forme:

`cv2.flip(img, n)`

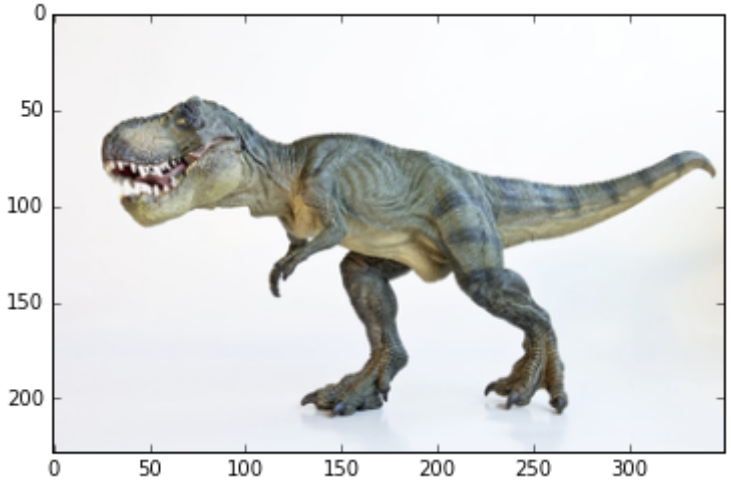
n prend les valeurs :

- 0 pour flipper autour de l'axe x (verticalement)
- 1 pour flipper autour de l'axe y (horizontalement)
- -1 pour flipper autour de l'axe x et de l'axe y

In [6]:

```
# Exemple:
img8 = cv2.imread('trex.png')

img8 = cv2.flip(img8, 1)
show(img8)
```



Rogner

Définition: le rognage permet de découper une partie de l'image et de l'afficher.

Il faut définir la partie que l'on veut afficher:
`tete = img[$y_1$:$y_2$, $x_1$:$x_2$]`

Puis on peut imprimer l'image directement.

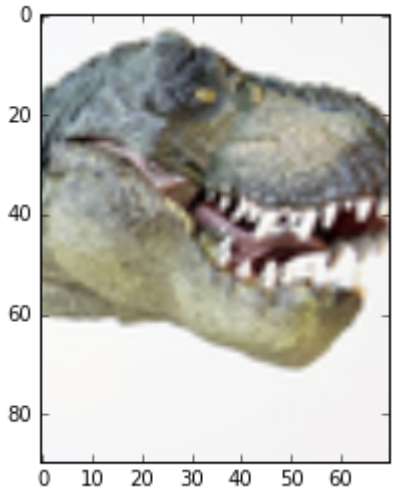
Si l'on veut colorier la partie découper on écrit:
`img[$y_1$:$y_2$, $x_1$:$x_2$] = (B,G,R)`

In [30]:

```
# Exemple:
img9 = cv2.imread('trex.png')

(x1, y1) = (250, 40)
(x2, y2) = (130, 320)

tete = img9[y1:y2, x1:x2]
show(tete)
```



Contraste

Définition: il existe deux fonction add ou subtract. Leur but est d'ajouter ou d'enlever un certain "nombre couleur" à chaque pixels de l'image.

La matrice OpenCV qui effectue le travail est de la forme:

`M = np.ones(img.shape, dtype = "uint8")*100`

`dtype = "uint8"` : définit le type de pixel, en loccurence on utilise des 8-bits
`*100` : represente le degré de couleur à enlever

Les deux fonctions s'appellent:

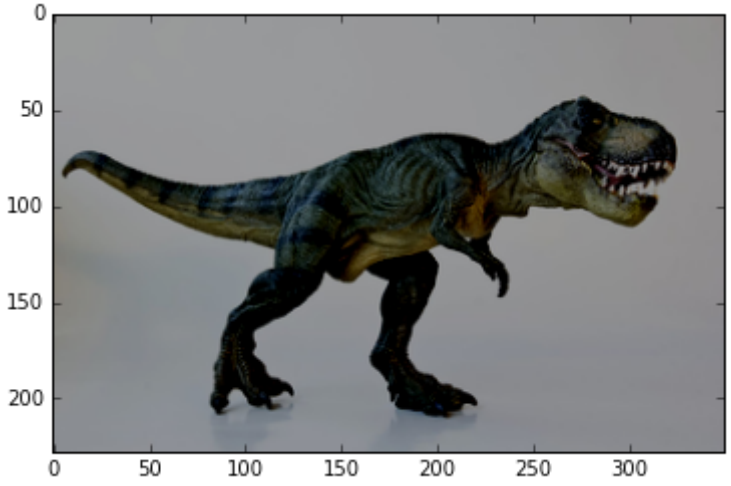
- `cv2.add(img, M)`
- `cv2.subtract(img, M)`

In [41]:

```
# Exemple:
img10 = cv2.imread('trex.png')

M = np.ones(img.shape, dtype = "uint8")*100
img10 = cv2.subtract(img10, M)
show(img10)

# Quand on utilise subtract, on enlève de la couleur et l'image paraît plus foncée;
# lorsqu'on met add l'image paraît plus claire.
```



Bitwise operations

Pour travailler avec les bitwise operations, il faut préalablement préparer les images en utilisant des pixels 8-bits:
`img = np.zeros((300,300), dtype = "uint8")`

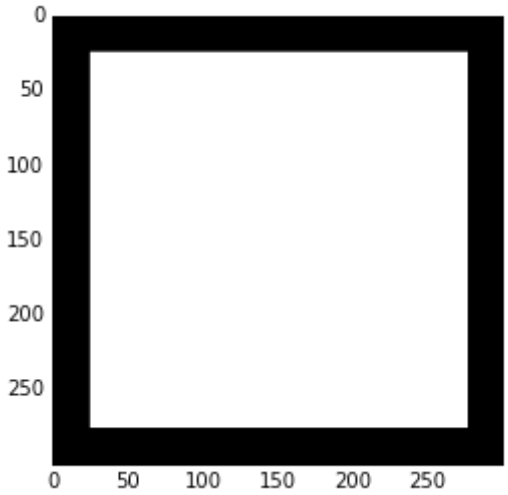
Ensuite on ajoute appelle les fonctions rectangle et cercle qu'on a vu précédemment. La différence c'est que dans ces fonctions, il faudra seulement mettre 0 ou 225 dans la parenthèse couleur.

In [48]:

```
# Préparation (1) du cadre de travail:
img01 = cv2.imread('trex.png')
img01 = np.zeros((300,300), dtype = "uint8")

img11 = cv2.rectangle(img01, (25,25), (275,275), 225, -1)

show(img11)
```

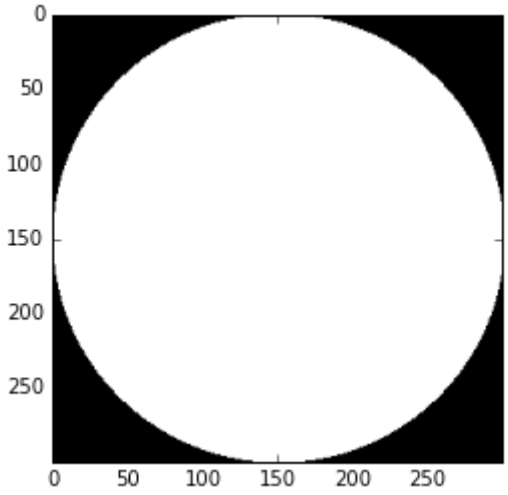


In [47]:

```
# Préparation (2) du cadre de travail:
img02 = cv2.imread('trex.png')
img02 = np.zeros((300,300), dtype = "uint8")

img12 = cv2.circle(img02, (150,150), 150, 225, -1)

show(img12)
```



Maintenant que nous avons deux cadre, nous allons voir les différentes opérations bitwise.

Operation 'And'

Définition: Cette opération superpose deux image et affiche en blanc uniquement les pixels qui sont communs aux deux images.

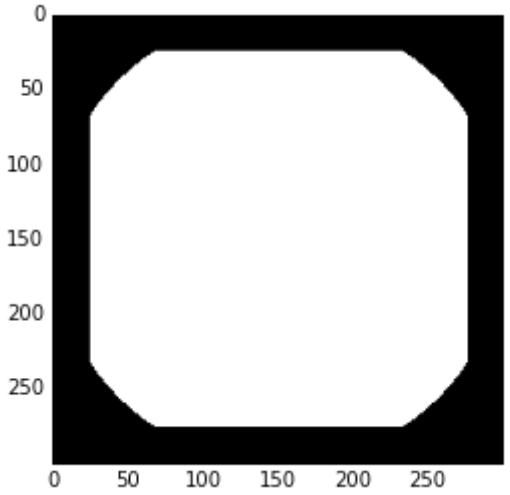
La fonction OpenCV de l'opératon And a la forme:

```
cv2.bitwise_and(img1, img2)
```

In [49]:

```
# Exemple:

bitwiseAnd = cv2.bitwise_and(img11, img12)
show(bitwiseAnd)
```



Operation 'Or'

Définition: Avec cette opération, l'image résultante affiche entièrement les deux formes superposées.

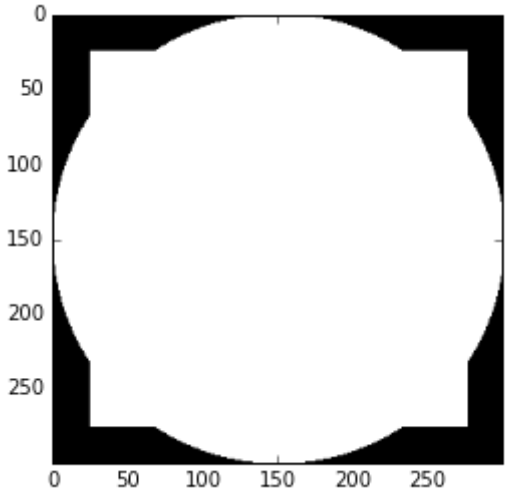
La fonction OpenCV de l'opération Or a la forme:

```
cv2.bitwise_or(img1,img2)
```

In [50]:

```
# Exemple:

bitwiseOr = cv2.bitwise_or(img11, img12)
show(bitwiseOr)
```



Operation 'Xor'

Définition: Cette opération consiste à afficher en noir les parties communes aux deux images et en blanc les parties uniques.

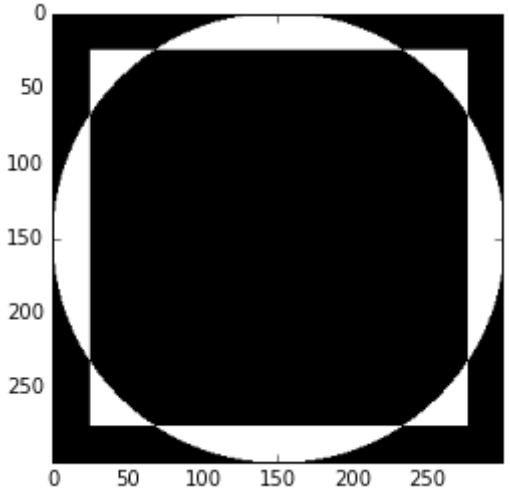
La fonction OpenCV de l'opération Xor a la forme:

```
cv2.bitwise_xor(img1,img2)
```

In [51]:

```
# Exemple:

bitwiseXor = cv2.bitwise_xor(img11, img12)
show(bitwiseXor)
```



Operation 'Not'

Définition: Cette opération fonctionne avec une seule image comme argument et inverse la valeur couleur. Si on a mis 225 (blanc) pour le rectangle, avec not il apparraîtra en 0 (noir).

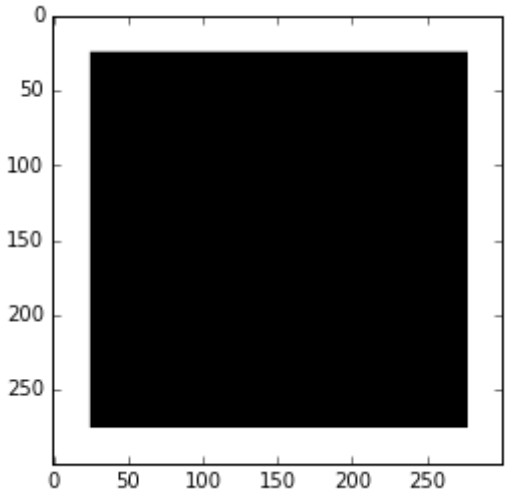
La fonction OpenCV de l'opération Not a la forme:

```
cv2.bitwise_not(img)
```

In [53]:

```
# Exemple:

bitwiseNot = cv2.bitwise_not(img11)
show(bitwiseNot)
```



Masquer une image

Définition: cette fonction permet de créer un cadre sur l'image. On peut définir ce cadre de tel sorte qu'une certaine partie de l'image soit visible et le reste soit noir. Pour réaliser un masque, on utlise les connaissances vu précédemment (rectangle, cercle, bitwise operations).

Il faut définir le masque (ex: masque rectangulaire):

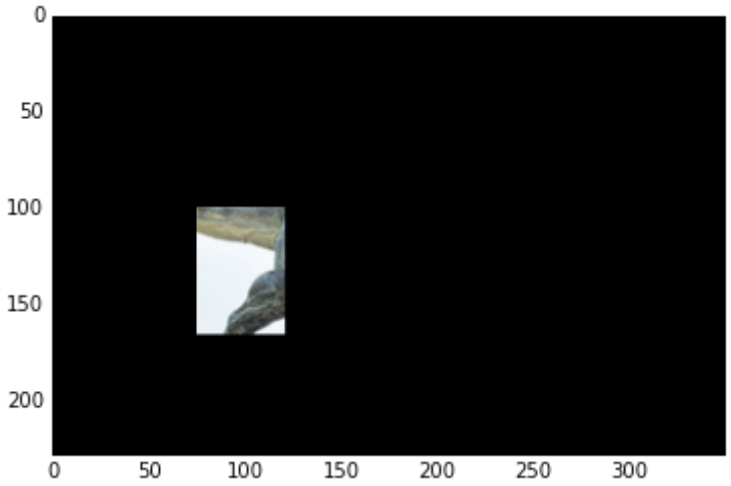
```
mask1 = np.zeros(img.shape[:2], dtype = 'uint8')
cv2.rectangle(mask1, ($x_1 , y_1$), ($x_2 , y_2$), 225, -1)
```

Ensuite on applique le masque à l'image avec l'opération `bitwise_and` .

In [12]:

```
# Exemple:
mask = np.zeros(img.shape[:2], dtype = 'uint8')
cv2.rectangle(mask, (75,100), (120,165), 225, -1)

masked = cv2.bitwise_and(img, img, mask = mask)
show(masked)
# Si lo'on veut créer un masque circulaire, il faut juste changer la fonction par cv2.c
```



Création de filtres

Définition: Cette fontion permet de mettre des filtres et faire resortir les nuances de couleur des images.

On utilise le système de couleur BGR et non RGB donc pour changer il suffit de faire:

```
(B,G,R) = cv2.split(img)
```

On définit:

```
zeros = np.zeros(img.shape[:2], dtype = 'uint8')
```

La fonction OpenCV pour mettre en place un filtre a la forme:

```
cv2.merge([zeros,G,zeros])
```

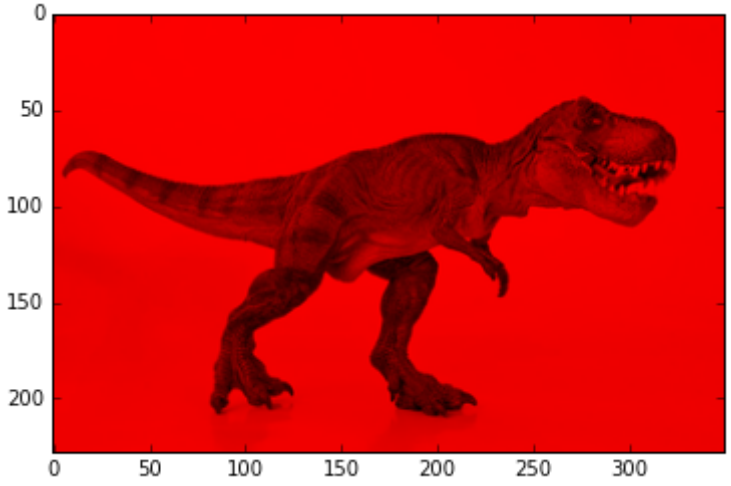
Dans ce cas, l'image sera verte.

In [16]:

```
# Exemple:
imgR = cv2.imread('trex.png')
(B,G,R) = cv2.split(imgR)
zeros = np.zeros(img.shape[:2], dtype = 'uint8')

imgR = cv2.merge([zeros,zeros,R])
show(imgR)

# Si on veut changer la couleur du filtre de l'image on modifie la position des zeros d
```



Espace de couleur

Définition: il existe plusieur 'espace de couleur': BGR (celui de base), RGB (que l'on connaît), le HSV, le gray (fait apparaître l'image en noir et blanc) et le lab. Ces 'espace de couleur' change l'apparence de l'image.

Fonctions pour passer dans les différents color spaces:

- cv2.COLOR_BGR2RGB => RGB
- cv2.COLOR_BGR2HSV => HSV
- cv2.COLOR_BGR2GRAY => gray
- cv2.COLOR_BGR2LAB => lab

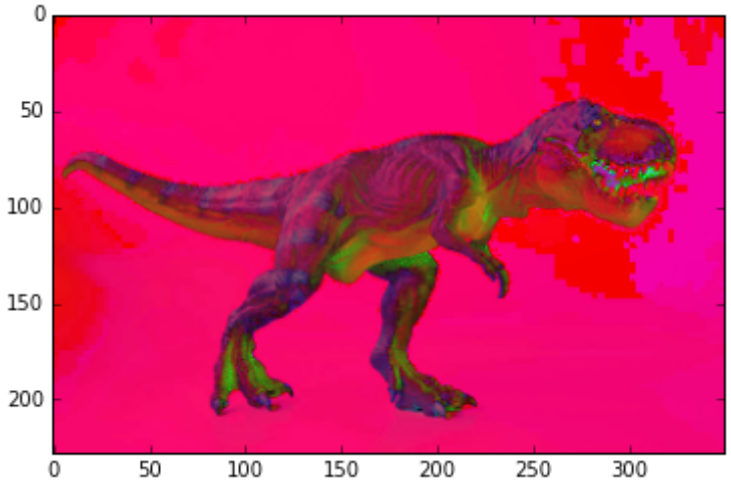
La fonction OpenCV pour changer l'espace de couleur a la forme:

```
cv2.cvtColor(img, fonction d'espace de couleur)
```

In [19]:

```
# Exemple:
img13 = cv2.imread('trex.png')

img13 = cv2.cvtColor(img13, cv2.COLOR_BGR2HSV)
show(img13)
```

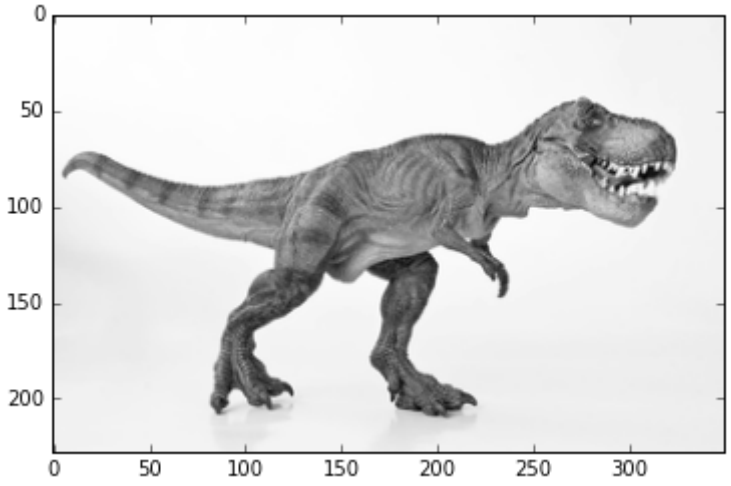


Histogrames

Pour illustrer ce chapitre sur les histogrames, nous allons utiliser une image en noir et blanc:

In [20]:

```
imgGray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
show(imgGray)
```



Utiliser OpenCV pour créer un histograme

La fonction OpenCV pour créer un histograme a la forme:

`cv2.calcHist([imgGray], channels, mask, histSize, ranges)`

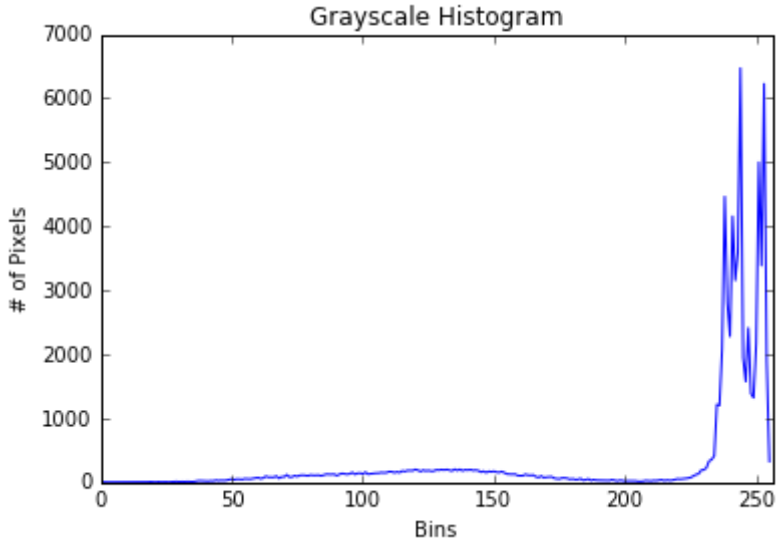
- `channels` : mettre la valeur 0 si l'image est grise et 1 ou 2 si elle est en couleur
- `mask` : s'il y en a un mettre le nom du mask, sinon mettre None

- histSize : prend les valeurs [256] ou [32,32,32], cela dépend de ce que l'on veut obtenir
- ranges : avec (R,G,B), il faut mettre [0,256]

In [22]:

```
# Exemple:
hist = cv2.calcHist([imgGray], [0], None, [256], [0,256])

histogram(hist, 'Grayscale Histogram') # J'ai utilisé directement la fonction histogram
```



Interprétation des histogrames

L'axe des x représente la couleur des pixels allant de 0 (noir) à 256 (blanc). Plus on est proche de l'ordonnée, plus les pixels sont foncés.
L'axe des y représente le nombre de pixels.

Sur notre graphique on peut donc en déduire qu'il y a un grand nombre de pixels de couleur (environ) blanche, ce qui est logique car le fond de l'image est blanc. Et il y a une légère bosse de la courbe autour de 130, ce qui montre qu'il y a un peu de pixels gris, que l'on retrouve nottament sur le corps su trex.

Images nettes et floues

Défintion: ces fonction permettent de rendre l'image flou ou non. Il existe quatre méthodes.

Méthode averaging

la fonction OpenCv pour floutter une image avec la méthode averaging a la forme:

```
cv2.blur(img,(n,n))
```

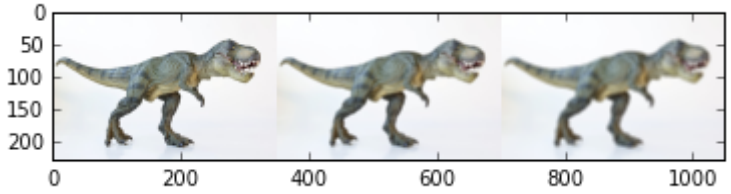
n est un nombre impair entier.

Pour voir les différences de floutage, on va utiliser une fonction permettant de coller verticallement ou horizontallement une suite d'image:

```
np.hstack([
    fonction1,
    fonction2,
    (...)
])
```

In [25]:

```
# Exemple:
blurred1 = np.hstack([
    cv2.blur(img,(3,3)), # Ne pas oublier les virgules à la fin de la fonction
    cv2.blur(img,(5,5)),
    cv2.blur(img,(7,7)),
])
show(blurred1)
```



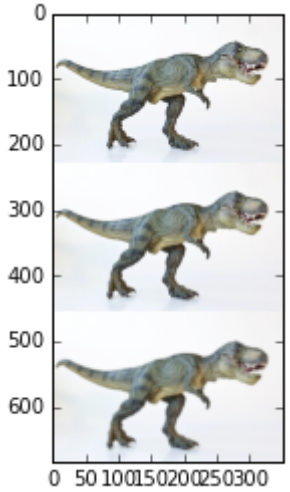
Méthode de Gauss

La fonction OpenCV pour floutter une image avec la méthode de Gauss a la forme:

```
cv2.GaussianBlur(img,(n,n),0)
```

In [27]:

```
# Exemple:
blurred2 = np.vstack([
    cv2.GaussianBlur(img,(3,3),0),
    cv2.GaussianBlur(img,(5,5),0),
    cv2.GaussianBlur(img,(7,7),0),
])
show(blurred2)
```



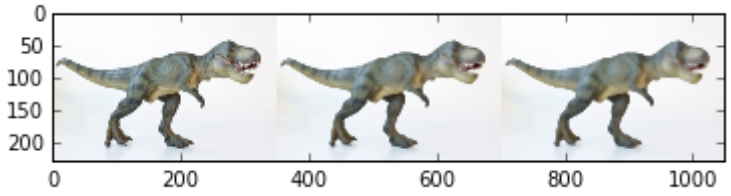
Méthode médiane

La fonction OpenCV pour floutter une image avec la méthode médiane a la forme:

```
cv2.medianBlur(img,n)
```

In [28]:

```
# Exemple:
blurred3 = np.hstack([
    cv2.medianBlur(img,3),
    cv2.medianBlur(img,5),
    cv2.medianBlur(img,7),
])
show(blurred3)
```



Méthode bilatérale

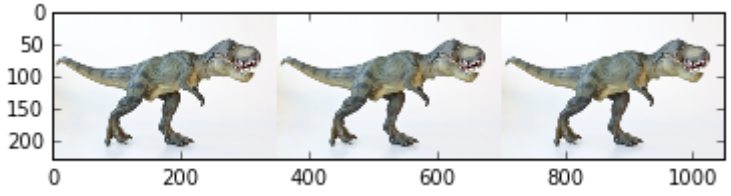
La fonction OpenCv pour floutter une image avec la méthode bilatérale a la forme:

```
cv2.bilateralFilter(img, n ,p, p)
```

si n = 5 p = 21, si n = 7 p = 31 (...)

In [29]:

```
# Exemple:
blurred4 = np.hstack([
    cv2.bilateralFilter(img,5,21,21),
    cv2.bilateralFilter(img,7,31,31),
    cv2.bilateralFilter(img,8,41,41),
])
show(blurred4)
```



Seuillage

Définuition: le seuillage est une fonction qui va déterminer selon une valeur (fixée par l'utilisateur), si les pixels seront coloriés en blanc ou en noir.

La fonction OpenCV pour le seuillage a la forme:

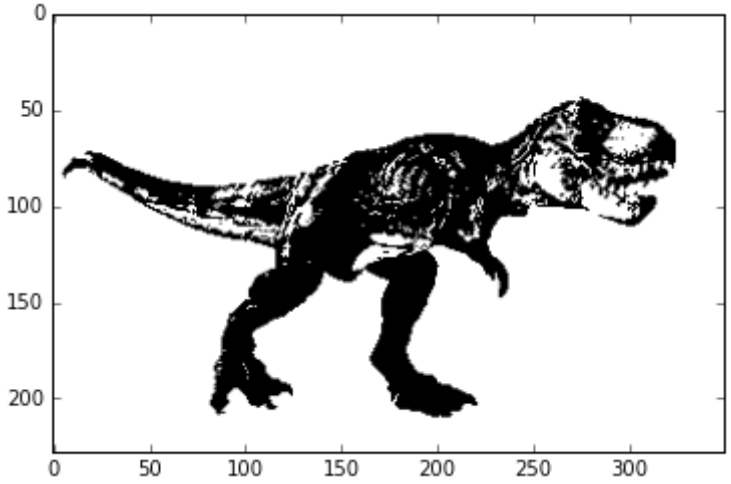
`cv2.threshold(img, a, b, cv2.THRESH_BINARY)`

`a` et `b` : sont les seuils fixés par l'utilisateur.
Si a=150 et b=225, alors tous les pixels < 150 seront en noir et tous les pixels > 150 seront blancs.

Pour faire l'inverse il suffite de metre `cv2.THRESH_BINARY_INV` dans la fonction.

In [37]:

```
# Exemple:
(T, thresh) = cv2.threshold(imgGray, 150, 225, cv2.THRESH_BINARY)
show(thresh)
```



Détécteur de formes

Définition: le détecteur de formes permet de dessiner les contours de certaines formes pour pouvoir les compter par la suite.

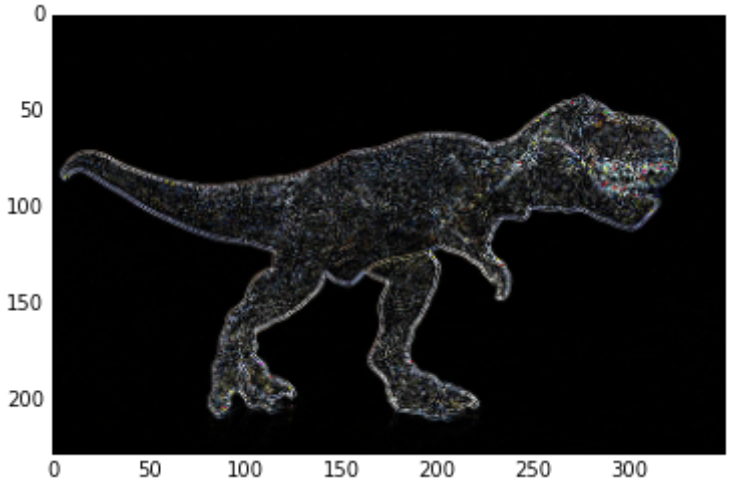
Méthode de Laplace

La fonction OpenCV pour détécter des formes avec la méthode de Laplace a la forme:

```
cv2.Laplacian(img, cv2.CV_64F)
np.uint8(np.absolute(img2))
```

In [38]:

```
# Exemple:
img14 = cv2.Laplacian(img, cv2.CV_64F)
img14 = np.uint8(np.absolute(img14))
show(img14)
```



Méthode Canny

Pour utiliser la méthode canny, il faut absolument que l'image soit floutée au minimum 5 fois auparavant.

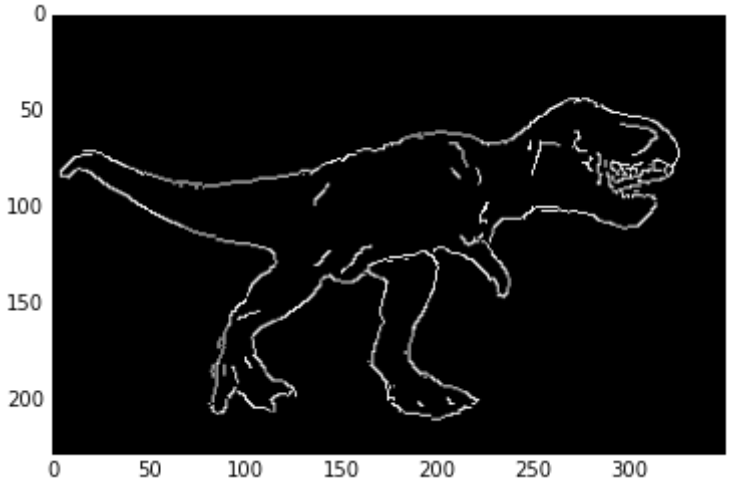
La fonction OpenCv pour détécter des formes avec la méthode Canny a la forme:

```
cv2.Canny(img, threshold1, threshold2)
```

In [52]:

```
# Exemple:
gray2 = cv2.blur(imgGray, (5,5))

canny = cv2.Canny(gray2, 100, 150)
show(canny)
```



Contours

Pour compter le nombre de formes grâce aux contours, on créer déjà un canny edge detector.

La fonction OpenCV pour compter les contours d'une image a la forme:

```
cv2.findContours(img.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

In [53]:

```
# Exemple:
(_, cnts, _) = cv2.findContours(canny.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLI
print(len(cnts))
```

```
58
```

Remarque: cette fonction n'est pas extrêment précise.