THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Basic Operations of PyParty

Adam Hughes

December 12, 2013

Contents

[1]George Washington University; Dept. of Physics; Condensed Matter Group (PI: Mark Reeves-reevesme@gwu.edu). Content claim excludes all references to to open source software, including iPython and its corresponding notebook and nbconvert utilities. Please direct inquiries to Adam Hughes (hugadams@gwmail.gwu.edu)

0.1 Imports/configuration

In [17]:
```
pylabl.rcParams['figure.figsize'] = 8, 5.5
import numpy as np
```

Part I

PyParty Tutorial

1 The Canvas Class

Canvas is the primary object that users will deal with in the pre-alpha version of pyparty. The canvas itself is a composite class, storing both an image (numpy.ndarray : *.image*) and a custom container for storing and manipulating particles (pyparty.ParticleManager : *.particles*).

To create a canvas, which is an image (ndarray) and a special container (ParticleManager) that stores, tracks and draws particles into the image (via array indexing).

In [18]:
```
from pyparty.tools.canvas import Canvas

c=Canvas()
print (type(c), type(c.image), type(c._particles))
```

```
(<class 'pyparty.tools.canvas.Canvas'>, <type 'numpy.ndarray'>, <class
'pyparty.tools.manager.ParticleManager'>)
```

The canvas tries to acts as an intuitive hybrid object for an image with particles, and selectives promotes the API of both images. For example, the public attribute *particles* hides the underlying *ParticleManager* instance, and instead returns particles as a list of tuples.

In [19]:
```
print type(c.image), type(c.particles)
```

```
<type 'numpy.ndarray'> <type 'list'>
```

The reason that *_particles* is a private method is because Canvas promotes its own ParticleManager api through *particles*, which be default, returns a list of particle names (see next section). Since canvas is a composite class between an ndarray and Particle manager, I decided that attribute lookup should defer to the image (eg):

In [20]:
```
print c.image.shape
print c.shape #
```

```
(512, 512, 3)
(512, 512, 3)
```

While slicing/indexing (see below) should defer to the particles, as the user could easily manipulate the image before and after applying particles.

2 Adding particles

Particles can be added to an image in a variety of ways.

2.1 Manually by passing parameters

First let's see what available particle types there are:

In [21]:
```
c._particles.available()
```

Out [21]:
```
('circle', 'dimer')
```

In [22]:
```
c.add('circle', name='centered_particle', radius=20, center=(255,256), col
c.add('circle', name='edge_particle', radius=100, center=(0,0), color=(0,1
c.add('circle', name='outside_particle', radius=20, center=(900,200), colo

named = c.particles
named
```

Out [22]:
```
['centered_particle', 'edge_particle', 'outside_particle']
```

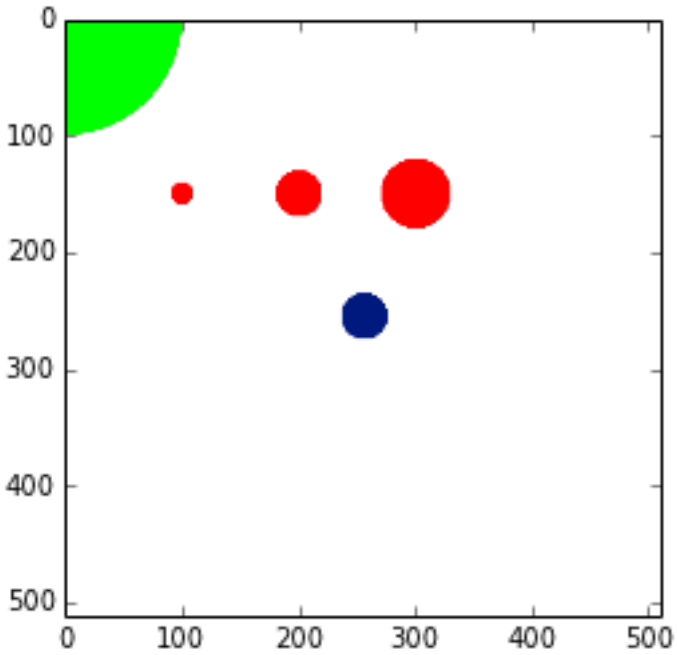
If we don't provide a name, they are automatically generated of the form (shape_index):

In [23]:
```
for idx, r in enumerate( (10, 20, 30) ):
c.add('circle', radius=r, center=(150, 100*(idx+1)), color=(1, 0, 0))

auto_named = [p for p in c.particles if p not in named]
auto_named
```

Out [23]:
```
['circle_4', 'circle_3', 'circle_5']
```

At any point, the image with drawn particles can be visualized with *show()*, a wrapper to *matplotlib.plt.imshow()*

In [24]:
```
c.show();
```



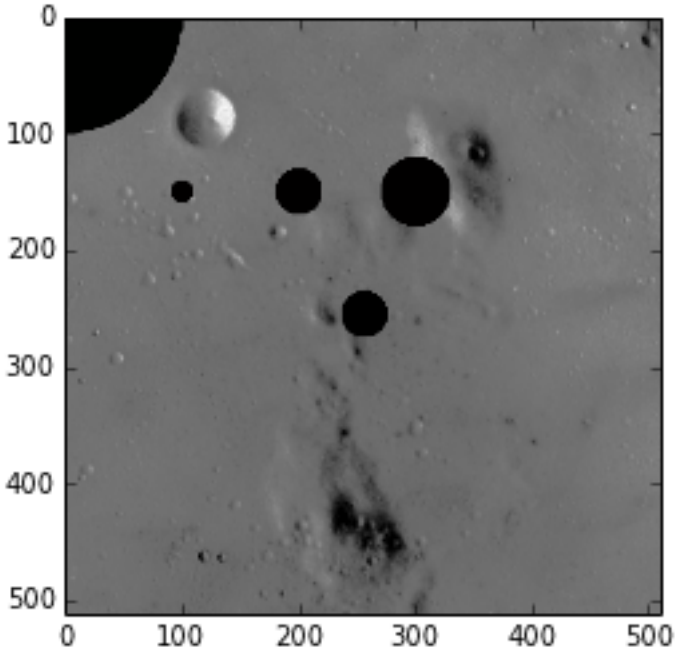
Notice that the large circles wrap around the corners. This is automatically done when image[rr, cc, :] is called...

3 Background images

The default background is a (1024 x 768 X 3) rgb array set to white. This was used implicitly above. We can actually pass any array (ie image) and the particles will be redrawn over it. In the next cell, we'll load an image from skimage's pre-packaged data.

In [25]:
```
from skimage.data import moon

c.background = moon()
c.show();
```

```
12-12 22:16:07 WARNING  pyparty.tools.canvas:  background color has
been converted (from grayscale to RGB)
```

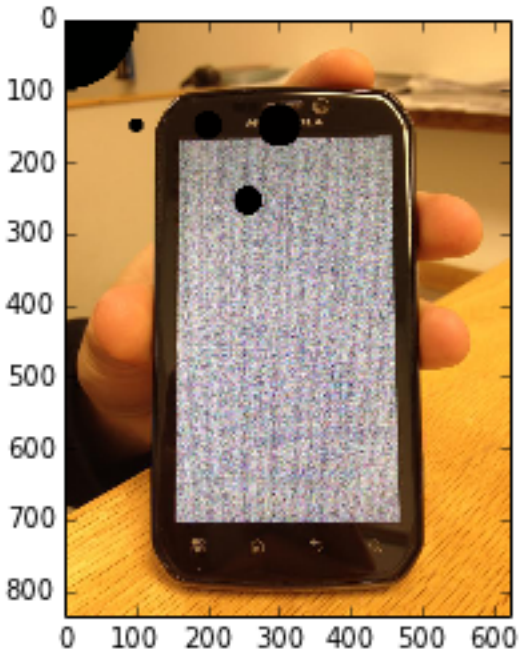


3.1 Image from harddrive/higher-resolution image:

Relative or absolute paths should be automatically detected. The image is returned by *load_background()*.

In [26]:
```
tip = c.load_background('~/Desktop/brokenphone.jpg') #Thanks Sprint...
c.show()
print 'newshape = ', c.shape
```

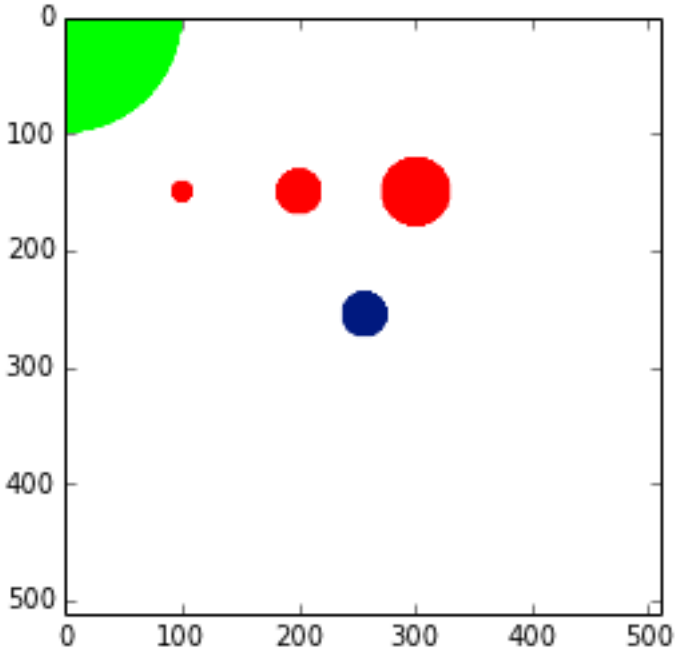
```
newshape =  (835, 626, 3)
```



The particles appear smaller because the resolution of this second image is larger. It is important to keep in mind that *particle coordinates are absolute in pixels*, so they Notice that the circles are applied to this image (top corner), **but are black(BUG?)**.

3.2 Clearing background

In [27]:
```
c.clear_background()
c.show();
```



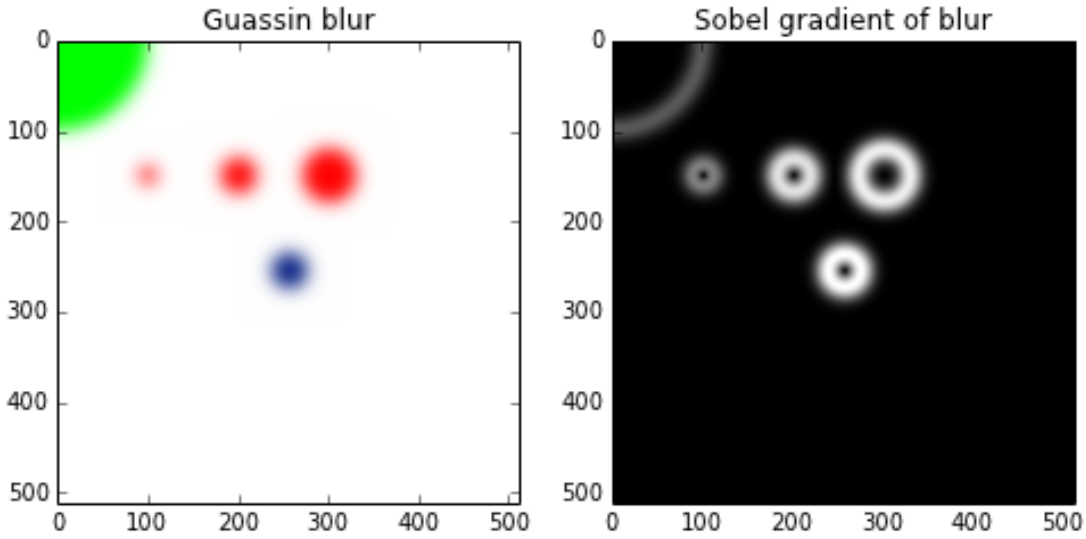
4 Integration with scikit image

The canvas object publicizes a few image attributes for convienence for example:

- shape
- ndim
- dtype

However, when using ndarray functions, pass the *.image* array directly.

In [28]:
```
from skimage.filter import gaussian_filter, sobel
from skimage.color import rgb2gray

fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(8,8))

c_filtered = gaussian_filter(c.image, 10)
ax1.imshow(c_filtered)
ax1.set_title('Guassin blur')

# Sobel requires gray image; we convert from rgb
c_sobel = sobel(rgb2gray(c_filtered))
ax2.imshow(c_sobel, plt.cm.gray)
ax2.set_title('Sobel gradient of blur');
```



ONE IMPORTANT CAVEAT: Since guassian_filter returns an array, c_filtered is no longer a *Canvas*. I may add some support later to return a *Canvas* object with it's image attribute overwritten; however, one should keep in mind that the underlying particles **are unchanged** by the guassian filter; only the final "drawn" image has been altered. It may therefore be misleading to add this feature because one would need to remain aware that these blurry particles are not representative of what is actually stored by the Canvas. In any case, this is at least a little more convienent than referring to c.image so often!

In [29]:
```
type(c), type(c_filtered)
```

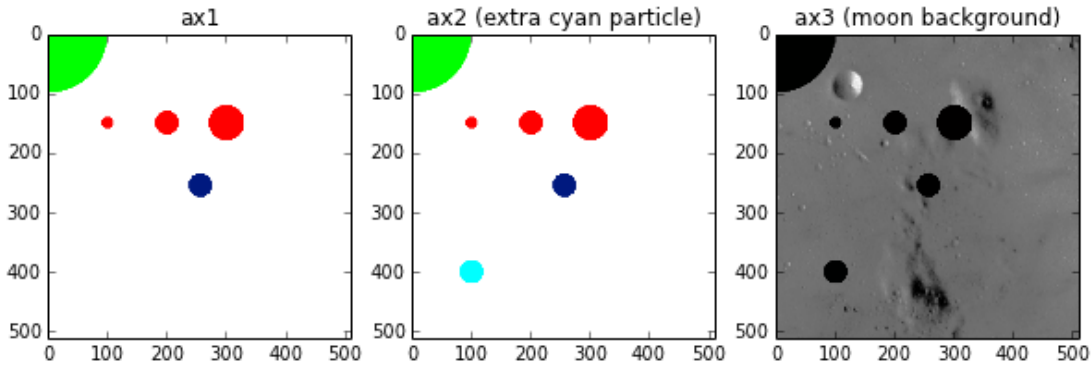
Out [29]:
```
(pyparty.tools.canvas.Canvas, numpy.ndarray)
```

5 Multiplots with matplotlib

Canvas has a *.show()* method that wraps imshow. This has been called already above to produce the output images. *.show()* also optionally accepts AxesSubPlot objects from matplotlib (these are returned by plt.subplots()). Thus, it's easy to create multiple plots

In [30]:
```
fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize = (10,10))

ax1.set_title('ax1')
c.show(ax1)

#Add a particle
c.add('circle', name='cyanparticle', radius=20, center=(400,100), color=((

ax2.set_title('ax2 (extra cyan particle)')
c.show(ax2)

#Reuse an old background for plot, then clear it
ax3.set_title('ax3 (moon background)')
c.background = moon()
c.show(ax3);
```

```
12-12 22:16:10 WARNING  pyparty.tools.canvas:  background color has
been converted (from grayscale to RGB)
```



5.1 Clearing the particles/objects

The important methods are:

- remove(index or name) : Removes a single particle from canvas #NOT YET IMPLELMENTED
- clear_background() : Removes all particles from the canvas
- clear_particles() : Resets the background image to the default
- clear_canvas() : Removes all particles **and** resets the background

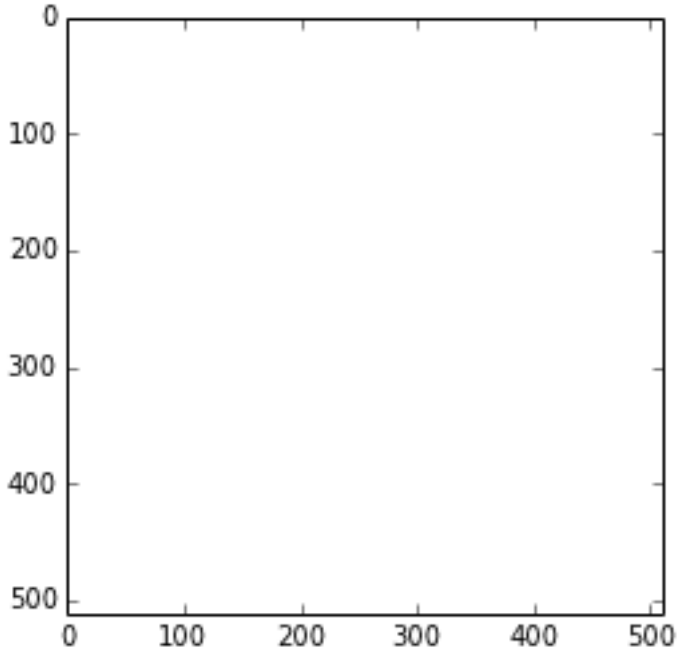
First, let's clear out the cyan particle that was just added. Since we named it "cyanparticle", this it is easy to access

In [31]:
```
print 'particles --->',c.particles
```

```
particles ---> ['centered_particle', 'circle_4', 'circle_3',
'cyanparticle', 'circle_5', 'edge_particle', 'outside_particle']
```

In [32]:
```
c.clear_canvas()
print 'particles --->', c.particles
c.show();
```

```
particles ---> []
```



6 Indexing/Slicing