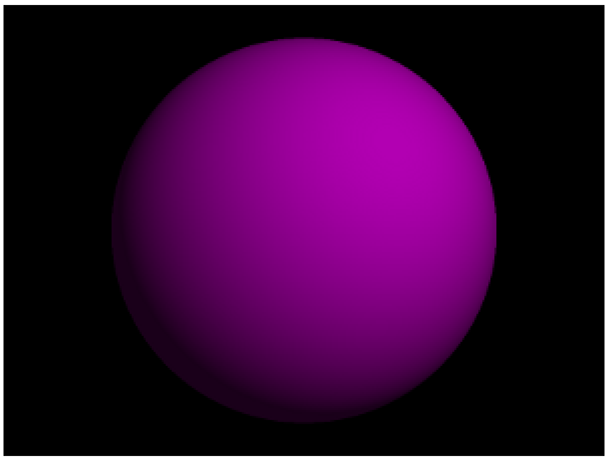
It may not be as good as Cinema4D or Pixar's RenderMan, but POV-Ray is free, open-source, and cross-platform. Rendering is launched from the terminal with `povray myscene.pov`, where `myscene.pov` contains the description of a 3D scene:

```
/* LET'S DRAW A PURPLE SPHERE ! */
camera { location <0, 2, -3> look_at <0, 1, 2>  }
light_source { <2, 4, -3> color <1, 1, 1> }
sphere { <0, 1, 2>, 2 texture{ pigment{ color <1, 0, 1> } } }
```



While POV-Ray has a very nice and sophisticated scene description language, I wanted to use it together with libraries from the Python world, so I wrote Vapory, a library to render POV-Ray scenes directly from Python, like this:

```
# LET'S DRAW A PURPLE SPHERE !
from vapory import *

camera = Camera( 'location', [0, 2, -3], 'look_at', [0, 1, 2] )
light = LightSource( [2, 4, -3], 'color', [1, 1, 1] )
sphere = Sphere( [0, 1, 2], 2, Texture( Pigment( 'color', [1, 0, 1] )))

scene = Scene( camera, objects= [light, sphere] )
scene.render("purple_sphere.png", width=400, height=300 )
```

This script simply generates a `scene.pov` file (hat tip this script by Simon Burton) and then sends the file to POV-Ray for rendering. Vapory can also pipe the resulting image back to Python, and has a few additional features to make it easy to use in an IPython Notebook.

Example 1: Basic animation with post-processing

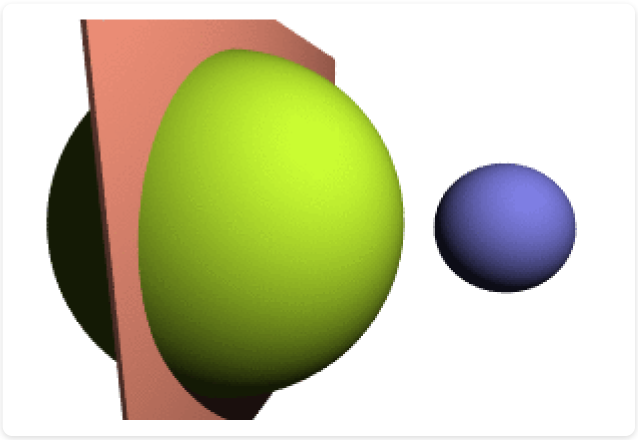
We first create a scene where the positions of the objects depend on the time :

```
from vapory import *

color = lambda col: Texture( Pigment( 'color', col))
```

```python

def scene(t):
    """ Returns the scene at time 't' (in seconds) """
    return Scene( Camera( 'location', [0, 2, -3], 'look_at',  [1, 1, 2] ),
              [ LightSource( [2, 4, -3], 'color', [1.5,1.5,1.5] ),
                Background( "color", [1,1,1] ),
                Sphere( [0, 1, 2] , 2,   color([.8, 1, .2])),
                Box( [-.8 + .5 * t, -1.5, -.5] , [-.75+.5*t, 3.5, 5], # <= t
                      color([1,.6,.5]), 'rotate', [0, 30, 0] ),
                Sphere( [ 3 - 2 * t , 1, 1.1] , .75,  color([.5, .5, .9]))])
```

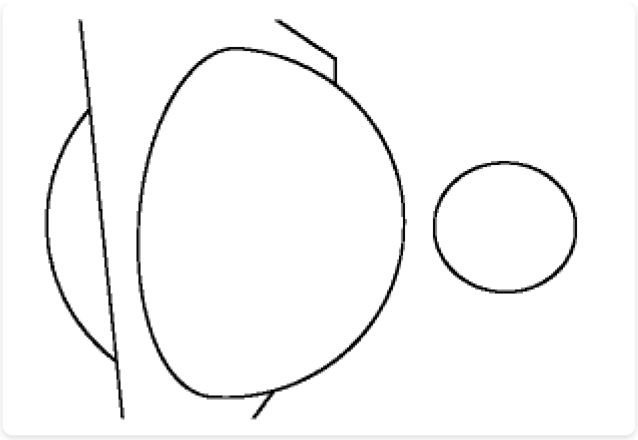
Then we animate this scene with MoviePy:

```python
from moviepy.editor import VideoClip

def make_frame(t):
    return scene(t).render(width = 300, height=200, antialiasing=0.001)

VideoClip(make_frame, duration=4).write_gif("anim.gif",fps=20)
```



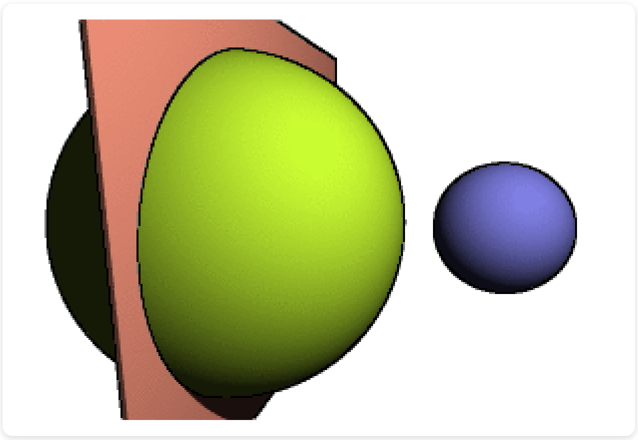
Note that one can also make basic animations directly with POV-Ray. But since we use Python we can use its image processing libraries for post-processing. As an example, let us use Scikit-image's sobel filter to obtain a nice geometry animation

```python
from skimage.filter import sobel
import numpy as np

def make_frame(t):
    # We will use "quality=1" so that shadows won't be rendered,
    # and double the rendering resolution to avoid pixelization.
    im= scene(t).render(width = 600, height=400,
                        antialiasing=0.001, quality=1)
    sobelized = np.array([sobel(1.0 * im[:,:,i]) for i in [0, 1, 2]])
    return np.dstack(3*[255*(sobelized.max(axis=0)==0)])

clip = VideoClip(make_frame, duration=4).resize(0.5)

clip.write_gif("anim_sobel.gif",fps=20)
```



The contours look pretty nice because POV-Ray uses exact formulas to render geometrical objects (contrary to libraries like ITK or OpenGL, which rely on triangular meshes). With a few more lines we can mix the two animations to create a cel-shading effect:

```python
from moviepy.editor import VideoFileClip
normal = VideoFileClip("anim.gif") # The first animation
sobelized = VideoFileClip("anim_sobel.gif") # The second animation
# We take the frame-by-frame minimum of the two animations
cel_shade = lambda gf, t: np.minimum(gf(t), sobelized.get_frame(t))
normal.fl(cel_shade).write_gif("cel_shaded.gif")
```

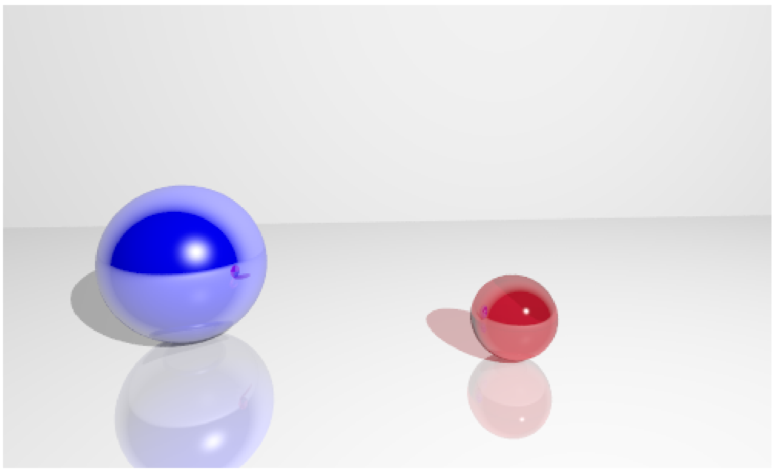


Example 2: Embedding a video in a 3D scene

Since we are playing around with MoviePy, let’s embed an actual movie in a 3D scene:

We start with a basic scene:

```python
from vapory import *

light = LightSource([10, 15, -20], [1.3, 1.3, 1.3])
wall = Plane([0, 0, 1], 20, Texture(Pigment('color', [1, 1, 1])))
ground = Plane( [0, 1, 0], 0,
                Texture( Pigment( 'color', [1, 1, 1]),
                         Finish( 'phong', 0.1,
                                 'reflection',0.4,
                                 'metallic', 0.3)))
sphere1 = Sphere([-4, 2, 2], 2.0, Pigment('color', [0, 0, 1]),
                                   Finish('phong', 0.8,
                                          'reflection', 0.5))
sphere2 =Sphere([4, 1, 0], 1.0, Texture('T_Ruby_Glass'),
                Interior('ior',2))

scene = Scene( Camera("location", [0, 5, -10], "look_at", [1, 3, 0]),
               objects = [ ground, wall, sphere1, sphere2, light],
               included=["glass.inc"] )
```



To this scene we will add a flat box (our *theater screen*), and for each frame of the movie we will make a PNG image file that will be used by POV-Ray as the texture of our flat box.

```python
from moviepy.video.io.ffmpeg_writer import ffmpeg_write_image

def embed_in_scene(image):

    ffmpeg_write_image("__temp__.png", image)
    image_ratio = 1.0*image.shape[1]/image.shape[0]
    screen = Box([0, 0, 0], [1, 1, 0], Texture(
                    Pigment( ImageMap('png', '"__temp__.png"', 'once')),
                    Finish('ambient', 1.2) ),
                 'scale', [10, 10/image_ratio,1],
                 'rotate', [0, 20, 0],
                 'translate', [-3, 1, 3])
    new_scene = scene.add_objects([screen])
    return new_scene.render(width=800, height=480, antialiasing=0.001)

clip = (VideoFileClip("bunny.mp4") # File containing the original video
        .subclip(23, 47) # cut between t=23 and 47 seconds
        .fl_image(embed_in_scene)  # <= The magic happens
        .fadein(1).fadeout(1)
        .audio_fadein(1).audio_fadeout(1))
clip.write_videofile("bunny2.mp4",bitrate='8000k')
```

This 25-seconds clip takes 150 minutes to generate (!!!) which may be due to the good resolution settings, numerous light reflexions in the balls and the ground, and the complex texture of the screen.

Example 3: A more complex scene

In this exemple we write “VAPORY” using 240 bricks:

First, we generate an image of the white-on-black text “VAPORY”. Many libraries can do that, here we use ImageMagick through MoviePy:

```
from moviepy.editor import TextClip

txtclip = TextClip("VAPORY", font="8BIT-WONDER-Nominal", kerning=2,
                   fontsize=8, bg_color='black', color='white')
txt_image = txtclip.get_frame(0)
```

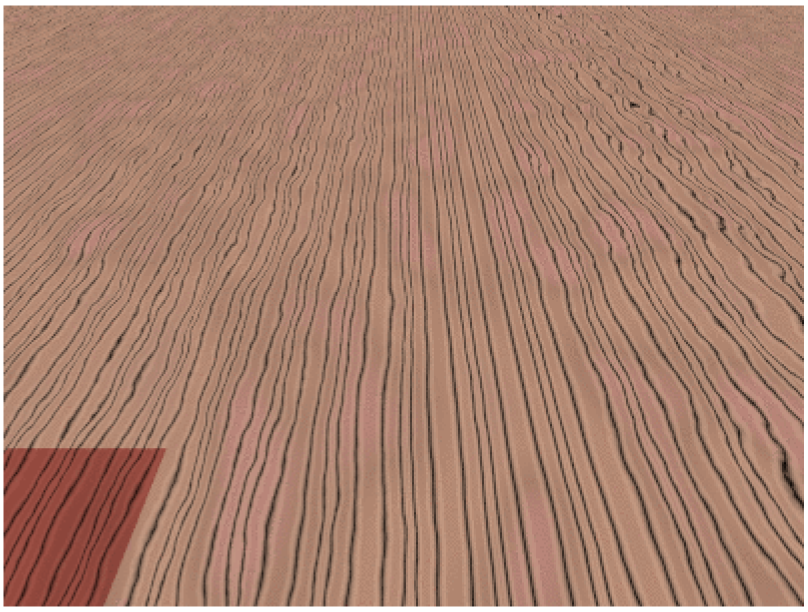
Here is the result:

We then get the coordinates of the non-black pixels is this image, and use them to place the bricks in the 3D scene, with small random variations around the depth-axis:

```
from vapory import *
import numpy as np

```

```python
# Compmute the coordinates of the 241 bricks

xx,yy = txt_image[:,:,0].nonzero()[::-1] # the non-black pixels
bricks_x = xx - 1.0 * (xx.max() + xx.min()) / 2
bricks_y = max(yy)  - yy + 1
bricks_z = np.random.normal(0, 0.08, len(xx))

# Generate / render the scene

bricks = [Box([x,y,z], [x+1,y+1,z-1], Texture("Sandalwood")) # The bricks
          for (x, y, z) in zip(bricks_xx, bricks_yy, bricks_zz)]
light = LightSource([-0, 50, -50], 'color', 1)
camera = Camera( 'location', [0, 5, -17], 'look_at', [0, 5, 0])

scene = Scene(camera, [light, Background("White")]+ boxes,
              included=["colors.inc", "textures.inc"])

scene.render("vapory.png", width=1000, height=240, antialiasing=0.001)
```

Example 4: Rendering a Physics simulation



Python as many nice scientific and engineering libraries that could benefit from a photorealistic rendering engine. Here I simulated the cube trajectories with PyODE (a Python binding of the physics engine ODE), and fed the results to Vapory and MoviePy for rendering and animation, all in a hundred lines.