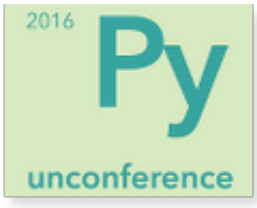


Tutorial: An introduction to pyRT ("Pirate") using the Jupyter Notebook

FHNW - University of Applied Sciences and Arts Northwestern Switzerland Martin Christen, martin.christen@fhnw.ch, @MartinChristen

pyRT is in development. Everything is (pre-)alpha!

```
pip install pyrt --upgrade
```

pyRT doesn't have any dependencies, but it is recommended to install the following modules:

```
pip install pillow --upgrade
pip install numpy --upgrade
pip install moviepy --upgrade
pip install jupyter –upgrade
```

Basically, it is up to you how to display images, pyRT only creates lists with RGB Values etc.

Source code is available on github: https://github.com/martinchristen/pyRT

The project is also on Twitter: @PythonRayTracer

Part 1: Using pyrt to Draw

pyRT also contains some functionatly to create images using Standard functions to draw points, lines, circle, rectangles. This has nothing todo with ray tracing, but it is fun and an easy way to get started to "Image Generators".

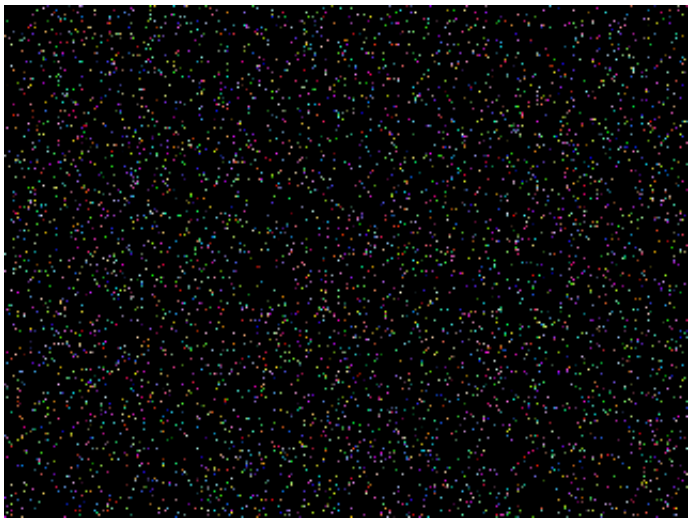
```
from pyrt.renderer import RGBImage
from pyrt.math import Vec2, Vec3
import numpy as np
import random
```

```
w = 320
h = 240
image = RGBImage(w, h)
```

```
for i in range(5000):
    position = Vec2(random.randint(0, w - 1), random.randint(0, h - 1))
    color = Vec3(random.uniform(0, 1), random.uniform(0, 1), random.uniform(0, 1))

    image.drawPoint(position, color)
```

Save image to file and display it

```
from IPython.display import Image
from PIL import Image as PImage

im = PImage.new("RGB", (w, h))
im.putdata(image.data)
im.save("rendering.png")
Image("rendering.png")
```

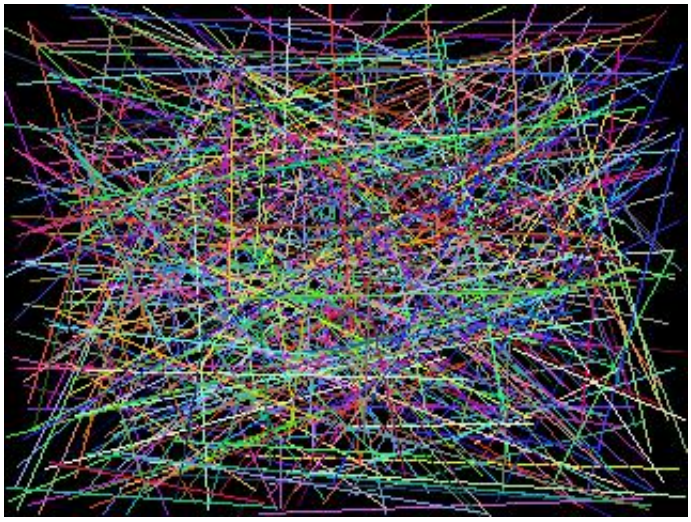


```
image = RGBImage(w, h)
```

```
for i in range(500):
    image.drawLine(Vec2(random.randint(0, w - 1), random.randint(0, h - 1)),
                   Vec2(random.randint(0, w - 1), random.randint(0, h - 1)),
                   Vec3(random.uniform(0, 1), random.uniform(0, 1), random.uniform
(0, 1)))
```

Output image without saving to file

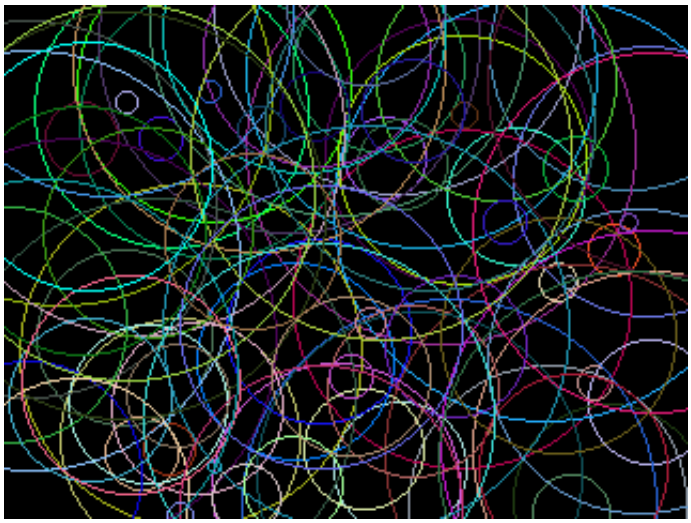
```
from IPython.core.display import HTML
import base64
from io import BytesIO

im = PImage.new("RGB", (w, h))
im.putdata(image.data)
buffer = BytesIO()
im.save(buffer, format="JPEG")
img_str = str(base64.b64encode(buffer.getvalue()), encoding="ascii")
HTML('<img src="' + img_str + '"></img>')
```



Personally I prefer this way of displaying an image, so let's create a display function for future images:

```
from IPython.core.display import HTML
import base64
from io import BytesIO

def display(imagedata, w, h):
    im = PImage.new("RGB", (w, h))
    im.putdata(imagedata)
    buffer = BytesIO()
    im.save(buffer, format="PNG")
    img_str = str(base64.b64encode(buffer.getvalue()), encoding="ascii")
    return HTML('<img src="' + img_str + '"></img>')
```

```
image = RGBImage(w, h)
for i in range(100):
    image.drawCircle(Vec2(random.randint(0, w - 1), random.randint(0, h - 1)),
                     random.randint(3, 100),
                     Vec3(random.uniform(0, 1), random.uniform(0, 1), random.unifo
rm(0, 1)))

display(image.data,w,h)
```



As a last example on how to "draw" directly, we use the "drawRectangle" method to - surprise - draw some rectangles.

```
image = RGBImage(w, h)

for i in range(100):
    image.drawRectangle(Vec2(random.randint(0, w - 1), random.randint(0, h - 1)),
                        random.randint(1, w / 2), random.randint(1, h / 2),
                        Vec3(random.uniform(0, 1), random.uniform(0, 1), random.un
iform(0, 1)))

display(image.data,w,h)
```

Part 2: An introduction to 3D Graphics

This part shows how we can draw in 3D. It introduces the submodules camera and geometry.

```
from pyrt.renderer import RGBImage
from pyrt.math import Vec2, Vec3
from pyrt.camera import PerspectiveCamera
from pyrt.geometry import Triangle, Vertex
```

let's create a 320x240 image again:

```
w = 320
h = 240
image = RGBImage(w, h)
```

Now we create a camera: the image plane has the same size like out output image (320x240) and we define a field of view of 60 degrees.

```
camera = PerspectiveCamera(w,h,60)
```

This camera has its origin in (0,0,0) and points along the z-axis. This is not really what we want. So we can set where it is positioned and where it looks at.

```
camera.setview(position, lookat, upvector)
```

position: camera position lookat: where it looks at upvector: for the orientation of the camera

```
camera.setView(Vec3(0,-10,0), Vec3(0,0,0), Vec3(0,0,1))
```

we can access the projection matrix and the view matrix using:

```
camera.projection
camera.view
```

These are 4x4 matrices.

```
print("Projection:")
print(camera.projection)
print("View:")
print(camera.view)
```

```
Projection:
[[1.2990381056766582, 0.0, 0.0, 0.0]
[0.0, 1.7320508075688776, 0.0, 0.0]
[0.0, 0.0, -1.0002000200020003, -0.20002000200020004]
[0.0, 0.0, -1.0, 0.0]]
View:
[[1.0, 0.0, -0.0, -0.0]
[0.0, 0.0, 1.0, -0.0]
[0.0, -1.0, 0.0, -10.0]
[0.0, 0.0, 0.0, 1.0]]
```

Thew view-projection matrix can be created by multiplying the two matrices. Please note that multiplications are done from right to left.

So we create the view-projection matrix using:

```
vp =  camera.projection * camera.view

print(vp)
```

```
[[1.2990381056766582, 0.0, 0.0, 0.0]
[0.0, 0.0, 1.7320508075688776, 0.0]
[0.0, 1.0002000200020003, 0.0, 9.801980198019804]
[0.0, 1.0, 0.0, 10.0]]
```

Now we create a triangle. Triangles consists of 3 Vertices. A Vertex can have attributes like position, color, normal, ...).

For this demo we only care about positions.

```
t = Triangle(Vertex(position=(-5, 1, 0)),
             Vertex(position=(0, 1, 5)),
             Vertex(position=(5, 1, 0)))
```

Now we multiply every vertex position of the triangle (t.a.position, t.b.position, t.c.position) with the view-projection matrix.

This results in normalized device coordinates (NDC) in the range (-1,-1,-1) to (1,1,1)

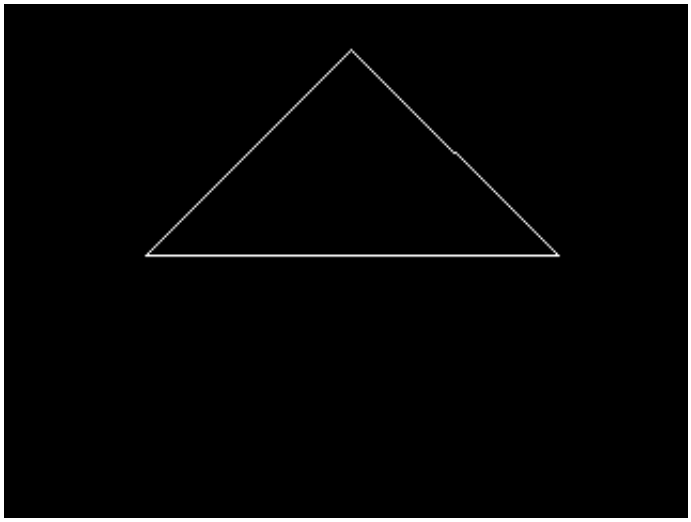
```
at = vp * t.a.position
bt = vp * t.b.position
ct = vp * t.c.position
```

The NDC are now transformed to image coordinates. The division by z is a perspective transformation.

```
a_screenpos = Vec2(int(w * 0.5*(at.x + 1.) / at.z), int(h * 0.5*(at.y + 1.)/ at.z)
)
b_screenpos = Vec2(int(w * 0.5*(bt.x + 1.) / bt.z), int(h * 0.5*(bt.y + 1.)/ at.z)
)
c_screenpos = Vec2(int(w * 0.5*(ct.x + 1.) / ct.z), int(h * 0.5*(ct.y + 1.)/ at.z))
```

And now we display the triangle by drawing the edges:

```
color = Vec3(1,1,1)
image.drawLine(a_screenpos, c_screenpos, color)
image.drawLine(c_screenpos, b_screenpos, color)
image.drawLine(b_screenpos, a_screenpos, color)

display(image.data,w,h)
```



Part 3: Ray Tracing

With this knowledge we know how 3D graphics basically work. Now we use ray-tracing to create the same triangle as in the last example.

First we import the requires (sub-)modules

```
from pyrt.math import *
from pyrt.scene import *
from pyrt.geometry import Triangle, Vertex
from pyrt.camera import PerspectiveCamera
from pyrt.renderer import SimpleRT
```

Then we create our camera

```
w = 320
h = 240

camera = PerspectiveCamera(w, h, 60)
camera.setView(Vec3(0,-10,0), Vec3(0,0,0), Vec3(0,0,1))
```

The next step is to create a scene. A scene consists of all objects you want to display. We just add a triangle to the scene.

```
scene = Scene()
scene.add(Triangle(Vertex(position=(-5, 1, 0)),
                   Vertex(position=(0, 1, 5)),
                   Vertex(position=(5, 1, 0))))
```

Now the scene has to know which camera we use for rendering, so we just set the camera:

```
scene.setCamera(camera)
```

Now we specify the raytracer. At the moment there is only one (reference) implementation of a raytracer, called **SimpleRT**

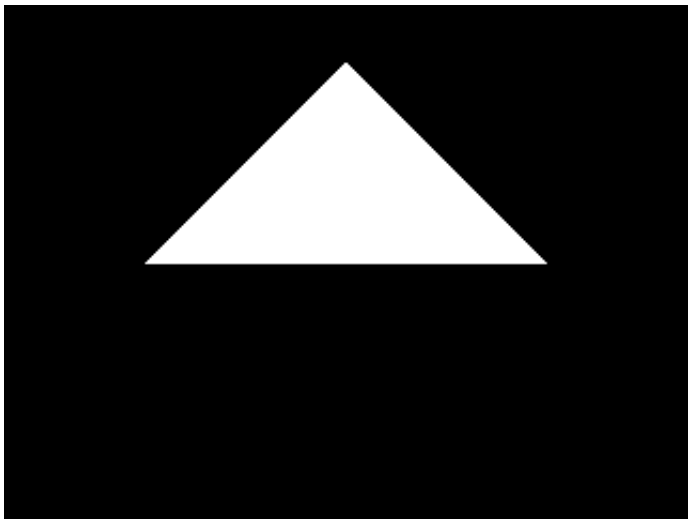
```
engine = SimpleRT()
```

```
# Creating Renderer: Simple Raytracer
```

Now we render the scene and display the result

```
imgdata = engine.render(scene)
display(imgdata,w,h)
```

```
# RENDER STATISTICS##################################
TIME FOR RENDERING: 2.3619461059570312s
NUMBER OF PRIMARY RAYS: 76800
NUMBER OF SECONDARY RAYS: 0
NUMBER OF SHADOW RAYS: 0
RAYS/s: 32515.559862396432
####################################################
```



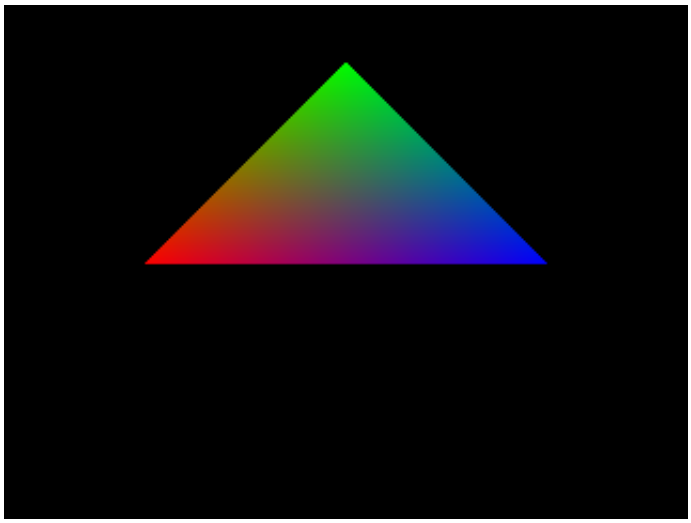
We create a new scene and this time we add some colors to the vertices:

```
scene = Scene()

scene.add(Triangle(Vertex(position=(-5, 1, 0), color=(1,0,0)),
                   Vertex(position=(0, 1, 5), color=(0,1,0)),
                   Vertex(position=(5, 1, 0), color=(0,0,1))))

scene.setCamera(camera)
```

and render again...

```
imgdata = engine.render(scene)
display(imgdata,w,h)
```

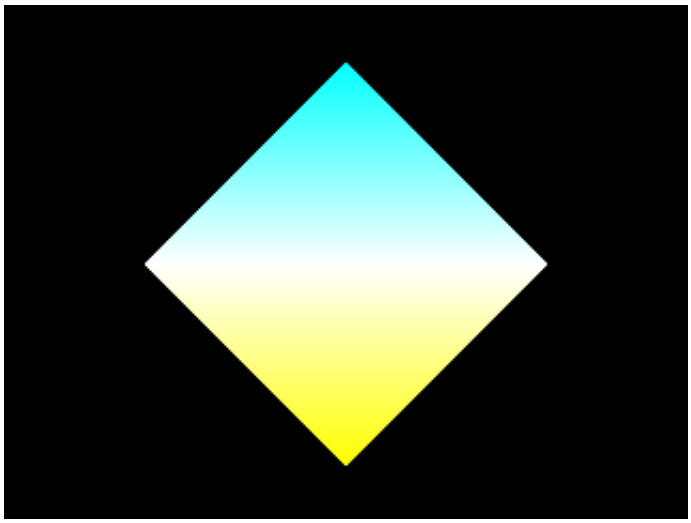
```
# RENDER STATISTICS#################################
TIME FOR RENDERING: 2.359083890914917s
NUMBER OF PRIMARY RAYS: 76800
NUMBER OF SECONDARY RAYS: 0
NUMBER OF SHADOW RAYS: 0
RAYS/s: 32555.01014430431
##################################################
```



We can also create a scene with 2 triangles and render it:

```
scene = Scene()

scene.add(Triangle(Vertex(position=(-5, 1, 0), color=(1,1,1)),
                   Vertex(position=(0, 1, 5), color=(0,1,1)),
                   Vertex(position=(5, 1, 0), color=(1,1,1))))

scene.add(Triangle(Vertex(position=(5, 1, 0), color=(1,1,1)),
                   Vertex(position=(0, 1, -5), color=(1,1,0)),
                   Vertex(position=(-5, 1, 0), color=(1,1,1))))

scene.setCamera(camera)

imgdata = engine.render(scene)
display(imgdata,w,h)
```

```
# RENDER STATISTICS##################################
TIME FOR RENDERING: 3.6194040775299072s
NUMBER OF PRIMARY RAYS: 76800
NUMBER OF SECONDARY RAYS: 0
NUMBER OF SHADOW RAYS: 0
RAYS/s: 21218.962667581676
#####################################################
```



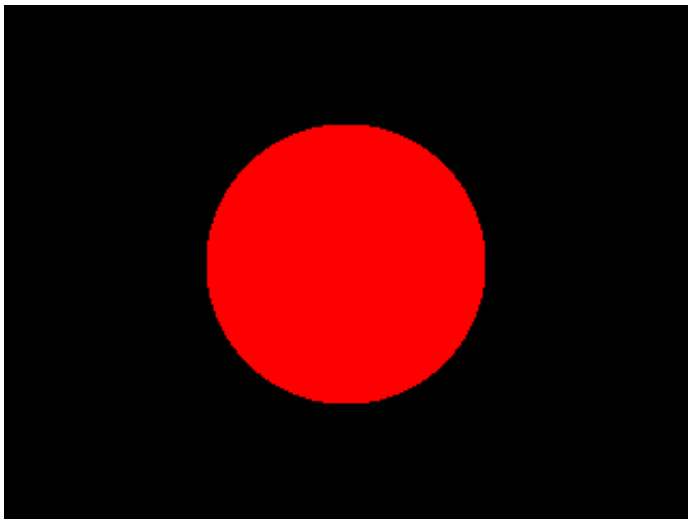
Instead of triangles we can also use spheres. Let's also look at materials. One material type is "PhongMaterial" where you can define the material of the object by specifying its color, its shinines, its reflectivity etc.

```
from pyrt.geometry import Sphere
from pyrt.material import PhongMaterial
```

```
scene = Scene()

scene.add(Sphere(center=Vec3(0.,0.,0.), radius=3., material=PhongMaterial(color=Ve
c3(1.,0.,0.))))

scene.setCamera(camera)
```

```
imgdata = engine.render(scene)
display(imgdata,w,h)
```

```
# RENDER STATISTICS################################
TIME FOR RENDERING: 2.091775894165039s
NUMBER OF PRIMARY RAYS: 76800
NUMBER OF SECONDARY RAYS: 0
NUMBER OF SHADOW RAYS: 0
RAYS/s: 36715.214193944885
#################################################
```

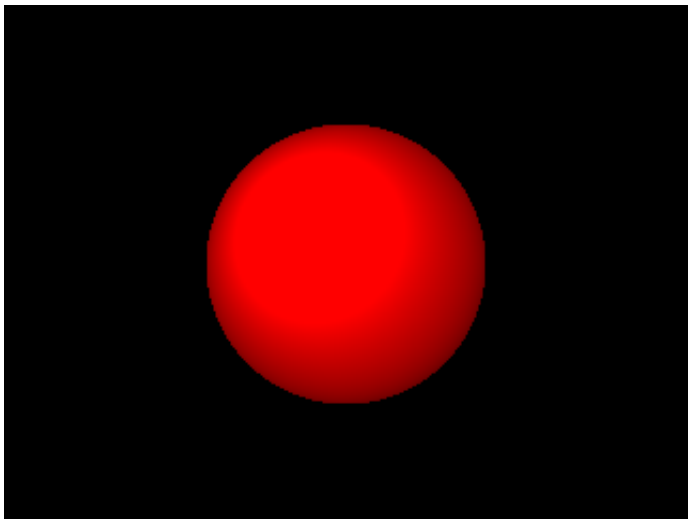


however, without light this object doesn't really look like a 3D-Object. So let's also add a point light. This light type is somewhat like a light bulb.

```
from pyrt.light import PointLight
```

```
scene = Scene()

scene.addLight(PointLight(Vec3(-1,-8,1)))
scene.add(Sphere(center=Vec3(0.,0.,0.), radius=3., material=PhongMaterial(color=Ve
c3(1.,0.,0.))))

scene.setCamera(camera)

imgdata = engine.render(scene)
display(imgdata,w,h)
```

```
# RENDER STATISTICS################################
TIME FOR RENDERING: 2.503495931625366s
NUMBER OF PRIMARY RAYS: 76800
NUMBER OF SECONDARY RAYS: 0
NUMBER OF SHADOW RAYS: 0
RAYS/s: 30677.101979605963
#################################################
```



Now we're ready to create a larger scene. Let's create 4 spheres on top of a plane created using two triangles. Every piece should have a different material, so let's create materials first.

```
floormaterial = PhongMaterial(color=Vec3(0.5,0.5,0.5))
sphere0material = PhongMaterial(color=Vec3(1.,0.,0.), reflectivity=0.5)
sphere1material = PhongMaterial(color=Vec3(0.,1.,0.), reflectivity=0.5)
sphere2material = PhongMaterial(color=Vec3(0.,0.,1.), reflectivity=0.5)
sphere3material = PhongMaterial(color=Vec3(1.,1.,0.), reflectivity=0.5)
```

Lets create another view, from more above looking to (0,0,0)

```
camera = PerspectiveCamera(w, h, 45)
camera.setView(Vec3(0.,-10.,10.), Vec3(0.,0.,0.), Vec3(0.,0.,1.))
```

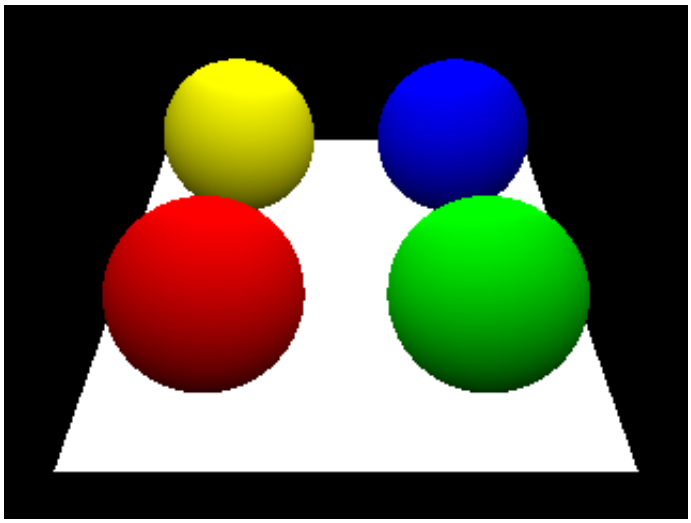
Now we create and add a light and geometries:

```
# Create a scene
scene = Scene()

# Add a light to the scene
scene.addLight(PointLight(Vec3(0,0,15)))


# Add "floor"
A = Vertex(position=(-5.0, -5.0, 0.0))
B = Vertex(position=( 5.0, -5.0, 0.0))
C = Vertex(position=( 5.0,  5.0, 0.0))
D = Vertex(position=(-5.0,  5.0, 0.0))

scene.add(Triangle(A,B,C, material=floormaterial))
scene.add(Triangle(A,C,D, material=floormaterial))

# Add 4 spheres
scene.add(Sphere(center=Vec3(-2.5,-2.5,1.75), radius=1.75, material=sphere0materia
l))
scene.add(Sphere(center=Vec3( 2.5,-2.5,1.75), radius=1.75, material=sphere1materia
l))
scene.add(Sphere(center=Vec3( 2.5, 2.5,1.75), radius=1.75, material=sphere2materia
l))
scene.add(Sphere(center=Vec3(-2.5, 2.5,1.75), radius=1.75, material=sphere3materia
l))

# Set the camera
scene.setCamera(camera)
```

now we are ready to render as usual:

```
imgdata = engine.render(scene)
display(imgdata,w,h)
```

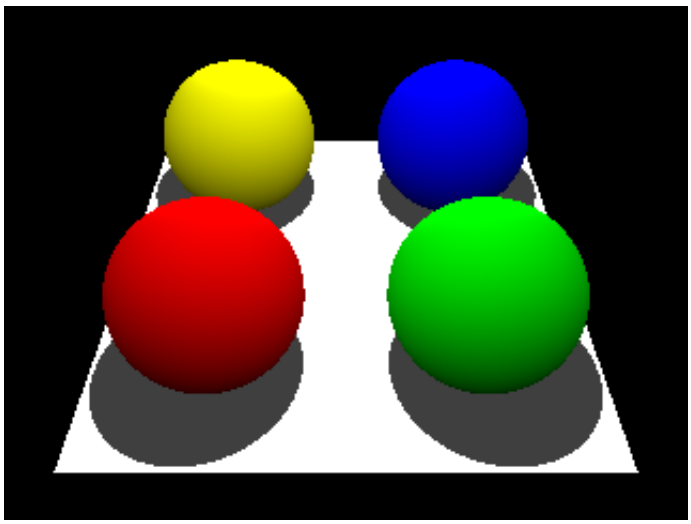
```
# RENDER STATISTICS#################################
TIME FOR RENDERING: 8.909627914428711s
NUMBER OF PRIMARY RAYS: 76800
NUMBER OF SECONDARY RAYS: 0
NUMBER OF SHADOW RAYS: 0
RAYS/s: 8619.888589918117
##################################################
```



This is quite a boring image. We can tell the renderer to support shadows:

```
engine = SimpleRT(shadow=True)
imgdata = engine.render(scene)
display(imgdata,w,h)
```

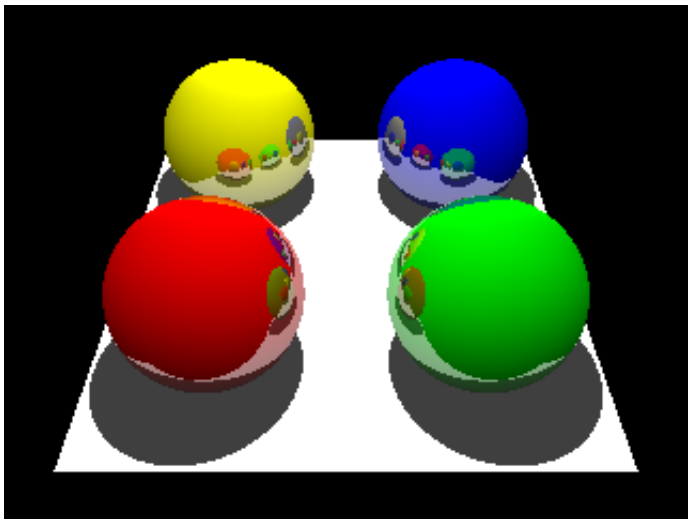
```
# Creating Renderer: Simple Raytracer
# Shadow Enabled
# RENDER STATISTICS################################
TIME FOR RENDERING: 11.061001062393188s
NUMBER OF PRIMARY RAYS: 76800
NUMBER OF SECONDARY RAYS: 0
NUMBER OF SHADOW RAYS: 55280
RAYS/s: 11941.053007314586
#################################################
```



And we can enable multiple iterations for rays. So a ray doesn't stop at an object if the material is reflecting.

```
engine = SimpleRT(shadow=True, iterations=3)
imgdata = engine.render(scene)
display(imgdata,w,h)
```

```
# Creating Renderer: Simple Raytracer
# Shadow Enabled
# Iterations: 3
# RENDER STATISTICS################################
TIME FOR RENDERING: 13.609395027160645s
NUMBER OF PRIMARY RAYS: 76800
NUMBER OF SECONDARY RAYS: 26456
NUMBER OF SHADOW RAYS: 62923
RAYS/s: 12210.608896894535
#################################################
```



```
```