Machine Learning for Rendering

Practice 1

2025-2026

Ricardo Marques
ricardo.marques@upf.edu
Universitat Pompeu Fabra – MIIS/EMAI

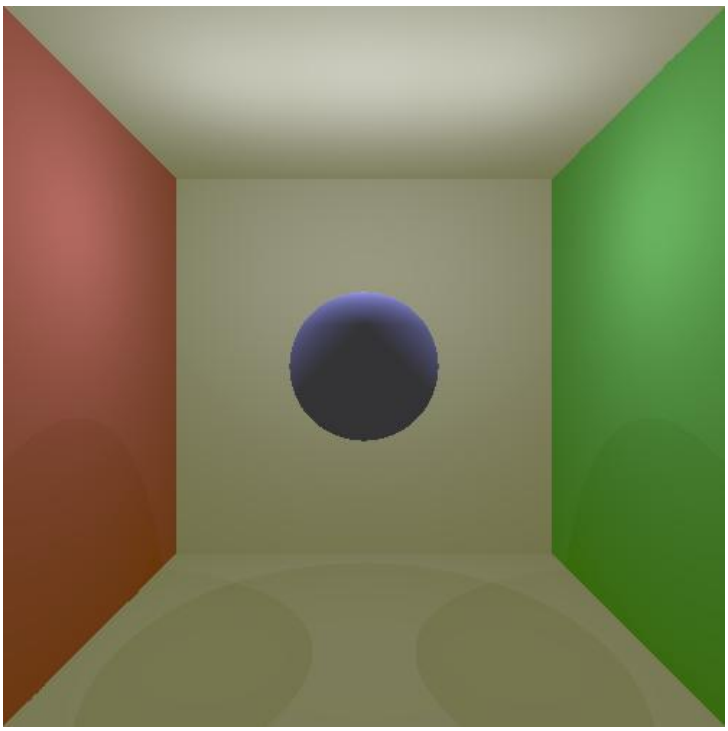

Figure 1. Cornell Box scene rendered using 3 point light sources and Lambertian materials.

Introduction to the PyRT Base Framework

In this first practice, you will have your first contact with the ray-tracing framework which will be used throughout the course to prototype photo-realistic rendering algorithms. At the end of the practice, you will have extended the PyRT base framework with Phong illumination-based rendering, yielding synthetic images such as the one shown in Figure 1 (above).

The objective of Practice 1 is two-fold:

1. To set-up your python working environment which you'll use for the remainder of the course – this is done in *Assignment 0* (1st week).
2. To get yourself familiar with the *PyRT* framework, as well as with the basic ray-tracing related concepts described in the theory class. To this end, you must implement the functionality of point light sources-based rendering in the original code – this is done in *Assignment 1* (2nd and 3rd weeks)

The *Python Ray-Tracer (PyRT)* is a set of classes and scripts written in Python specifically designed for the *Machine Learning for Rendering* (MLCG) course. The base framework is composed of four main files:

1. *PyRT_Core.py*: contains the main classes of the ray-tracer, such as the scene class, the camera class, the material classes or the primitive (i.e., object) classes;
2. *PyRT_Common.py*: contains a set of utility functions and classes, mostly related to the definition of useful data-types (*RBG_Color*, *Vector3D*, etc.), geometric operations and useful mathematical functions that we will use throughout the course;
3. *PyRT_Integrators.py*: this file contains the definition of the abstract base class (ABC) Integrator, responsible for synthesizing an image of a given scene;
4. *AppRenderer.py*: contains a script where the scene is created, rendered, saved to file, and shown on the screen.

Assignments

The provided base code of *PyRT*, however, is not complete. A set of functions required to implement rendering based on the Phong illumination model, such as the main rendering loop, or computing the intersection of a ray with a scene, are lacking. In the following, you will be provided with a sequence of assignments which will drive you toward a complete implementation of rendering based of the Phong illumination model.

Assignment 0: Set-up the python environment, the IDE and run the base code

Install Python. Open a terminal. Type "python" and ensure that you have a recent version of python installed in your computer (≥3.1).

Virtual Environments in Python. Virtual environments are a very useful tool for maintaining different python projects with different dependencies. Virtual environments allow managing dependencies separately for each project, preventing conflicts and maintaining cleaner setups. If you want to know more about virtual environments in python, check this web page. In this course, we will use *venv* to create python virtual environments – in principle you do not have to install it as it is part of all standard python distributions.

Create a Virtual Environment. Create a folder where you will keep the code for MLR – call it, for example, "2026_MLR_Code". To create the virtual environment, type:

python -m venv C:\[my_path]\2026_MLR\venv_MLR

where *[my_path]* represents the path to the *2026_MLR* folder which you just created, and *venv_MLR* is the actual folder where the virtual environment has just been created. If you navigate into the *venv_MLR* folder using a file explorer, you should see e folders, among which a folder named *Scripts*.

Activate and Deactivate the Virtual Environment. Now that your virtual environment is created, you can activate it by typing the full path to its activation script into a terminal:

C:\[my_path]\2026_MLR\venv_MLR\Scripts\Activate.ps1

It is possible that you get an error message stating that scripts are not activated. If this is the case, then you must run the following command:

Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser

This will enable scripts' execution within your terminal. Once this is doneand your virtual environment is activated, you should see something similar to this in your terminal:

```
PS C:\Users\u126915> C:/Users/u126915/Workspace/2026_MLR_code/venv_MLR/Scripts/Activate.ps1
(venv_MLR) PS C:\Users\u126915>
```

Finally, to deactivate the virtual environment just type deactivate in the terminal.

Set-up the IDE. Go to the Campus Global, enter in the course's page and download the base code for Assignment 1. Place the folder inside the *2026_MLR* folder which you have previously created. Install VSCode (not the standard Visual Studio!). Open VSCode. From within VSCode, open the folder with the base code for Assignment 1 – eventually accepting to install the python tools if suggested by VSCode.

Activating the Virtual Environment from VSCode. Click in the top bar and type:

>Select Interpreter

Once you've selected this option, then click on *Enter Interpreter Path* and select the file:

C:\[my_path]\2026_MLR\venv_MLR\Scripts\python.exe

You should now be able to execute the script in *AppRenderer.py* from within VSCode by clicking on the play button.

Installing Required Python Packages. You may need to install some python packages in order to run the *AppRenderer.py* script. For example, OpenCV (cv2) and matplot lib can be installed by typing the following into the VSCode terminal:

pip install opencv-python
pip install matplotlib

After a successful execution, a window should pop-up with a green image. The same image should be saved in the folder *out*.

Assignment 1: Phong – An Empiric Approach to Image Synthesis

Throughout this first assignment, you will mostly work on 2 classes: the *Integrator* class (and its children) and *Scene* class. An *Integrator* is the object in charge of generating an image of a given scene, using a particular technique. Each integrator implements a different rendering technique. Take a first look at the base class and remark that it has a scene attribute (declared in the constructor), and *a render()* method which, given the scene, will produce an image. As regards the *Scene* class, it is used to hold the main components of our 3D virtual scene: a camera which specifies how the scene is seen; the list of objects (aka primitives) which compose the scene; a list of point light sources as well as an ambient term, which you'll only use in this practice; the image to where the scene is rendered; and an environment map texture. You can execute the renderer by running the script *AppRenderer.py*.

Assignment 1.1: Implement the render function (Integrator base class)

Description: The goal of this assignment is to make you familiar with the main rendering loop, which can be found in the *render()* function of the *Integrator* abstract base class (ABC). The rendering loop traverses all the pixels of the image, and assigns a color value to them. Currently, it just assigns the green color to all of them. However, **you should change this behavior such that the color of each pixel is given by the pixel x and y coordinates**, such that:

$$color = RGBColor\left(\frac{x}{width}, \frac{y}{height}, 0\right),$$

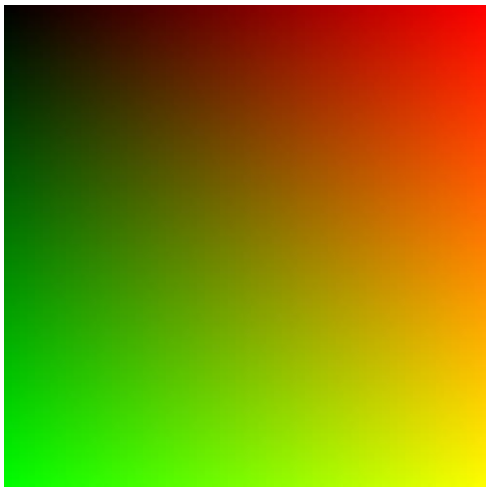

Figure 2. Expected result for Assignment 1.1.

where the *width* and the *height* correspond to the image resolution, and are fields of the *camera* object (named *cam* in the code). Note that, to set the value (i.e., the color) of a pixel, one must call the *set_pixel()* method of the scene object.

Validation: You can validate your implementation by comparing your result to that of Figure 2. If you have a different result, try to explain interpret the colors of your output image, and use it to correct the mistake in your implementation.

Assignment 1.2: Implement the Intersection Integrator

Description: The goal of the second assignment is to use PyRT to produce an image where pixels are either colored in red, or in black. Given a ray with origin in the camera position and passing through the center of a given pixel, the pixel will be colored in red if the ray intersects any object of the scene, and black otherwise. You can see an example of such a result in Figure 3.

The first step to complete this assignment is to implement the *any_hit()* function which you can find in the Scene class (PyRT_Core.py). Given a ray, this function should return true if the ray intersects any object of the scene, and false otherwise. To implement this functionality, you should check the following structures/classes of PyRT:

- The constructor (that is, the *__init__()* method) of the Scene class, where the Scene class attributes are defined. Note that a scene contains a list of objects, which you will need to manipulate in this assignment;
- The class *HitData* (*PyRT_Core.py*), which is used to store the information of a ray-object intersection test. Note the presence of a field named *has_hit,* which will be of particular interest to this assignment.
- The *intersect()* method which is implemented for all sub-classes of the abstract base class *Primitive*. **You do not have to understand the details of these computations. However, you should understand, in general terms, how this function behaves**: given a ray, this method returns a *HitData* object which contains the result of the intersection of that same ray with the 3D object (aka primitive) from which the *intersect()* method was called. If the ray intersects the object, then the intersection-related data will be stored in the *HitData* object returned by the *intersect()* method. If there is no

intersection, then the only meaningful field of the returned *HitData* object is *has_hit*, which is set to *False*.

Once you have the *any_hit()* method, you must complete the implementation of the *Intersection* integrator. As all integrators in *PyRT*, this integrator is defined as a child of the abstract base class *Integrator*, which makes the implementation of a *compute_color()* method compulsory. The *compute_color()* is probably the most important method with which you will deal with in *PyRT*. It is responsible for returning the color (*RGBColor* type) that is propagated along a given ray. In the case of the *Intersection* integrator, the *compute_color()* method returns red if there exists an intersection of the ray with the scene, and black otherwise.

Finally, to put it all together, you must use the *compute_color()* method to assign a color to each pixel of the image, by finalizing the rendering loop started in Assignment 1.1. To this end, for each pixel, you will need to set-up a ray with origin at the camera position (which is always $(0,0,0)$ in *PyRT*) and with a direction *d* such that the ray passes exactly through the center of the pixel. The direction *d* can be easily obtained using the *get_direction()* method of the Camera class, which receives as input the (x, y) coordinates of the desired pixel. Do not forget to replace the *LazyIntegrator* used in the previous assignment by your new *IntersectionIntegrator* in the *AppRenderer.py* script.

Figure 3. Expected result for the sphere scene rendered with the Intersection integrator.

Validation: You can validate your implementation by comparing your output with that of Figure 3. In case of differences, you should probably verify that you ray set-up is correct, as well as the exactness of the *any_hit()* implementation.

Assignment 1.3: Implement the Depth and Normal Integrators

Description: This assignment is about further manipulating and visualizing the intersection information present in the *HitData* structure. In contrast with the previous assignment, where we were only interested in determining whether or not a ray would hit any object of the scene, in this assignment we are interested in calculating the closest intersection along a ray. You must develop two distinct integrators: a *DepthIntegrator*, which renders an image such that the pixel color depends on the distance traveled by the ray till the closest intersection is detected; and a *NormalIntegrator*, which colors the pixels depending on the value of the vector perpendicular to the surface where lies the closest intersection point (we will refer to this vector as normal). In case there is no intersection detected, both integrators should pain the corresponding pixel in black. An example of the desired result is shown in Figure 4.

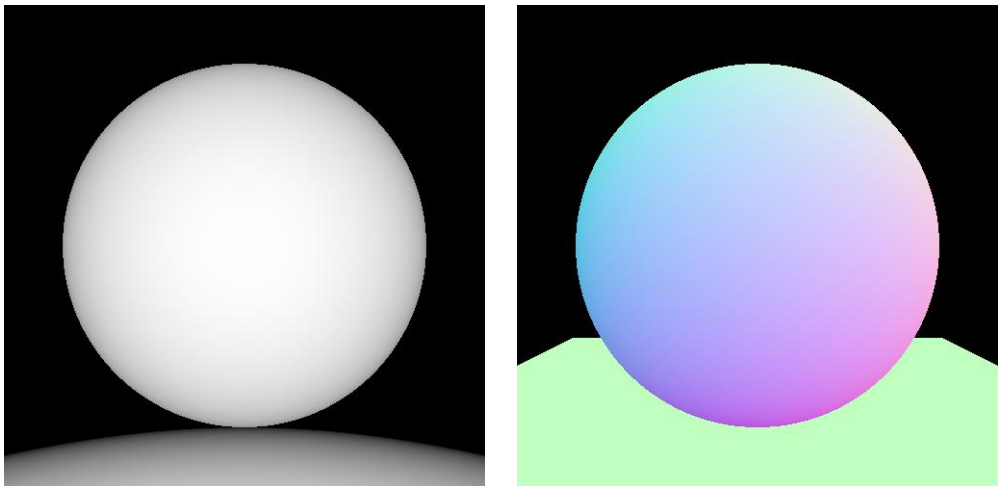

Figure 4. Left: Expected output for the sphere scene rendered using the DepthIntegrator with a max_depth of 5. Right: expected output for the same scene when using the normal integrator.

To implement these integrators, you must resort to the closest_hit() method of the scene class. Take a look at it and understand how it works. Once you know how to properly use this method to get the closest intersection along a given ray, you should implement the compute_color() methods of the Depth and Normal integrators. To convert a normal (Vector3D) to a color (RGBColor) you must take into consideration that the color components cannot be negative. Given a normal n, a color c encoding the normal vector can be computed as:

$$c = \frac{n + (1,1,1)}{2}$$

As regards coloring the pixel based on depth, the principle is to use gray-scale values (meaning that all rgb color components will always have the same value) to encode the distance. Pixels with intersection points closer to the viewer should have a "whiter" value than those which are further away. This effect can be achieved by applying the following equation:

$$c_i = \max\left(1 - \frac{hit\_distance}{max\_depth}, 0\right),$$

where c_i is one of the three color components (r, g and b), *hit_distance* is the field of the *hit_data* object eventually filled by the *closest_hit()* method, and *max_depth* is a parameter specified to the constructor of the *DepthIntegrator*.

Validation: To validate your implementation, simply compare your results with those show in Figure 4. Eventual differences could be explained by bugs in the *compute_color()* methods.

Assignment 1.4: Implement the Phong Integrator

Description. In the theory slides, you can find a detailed description of the Phong Illumination Model. Revise these slides and, once you understand all the details of the model, implement a *PhongIntegrator* in the *PyRT* framework. At this stage, you should be sufficiently familiar with the framework so that you can implement the integrator without requiring detailed guidance. However, there are a few minor points which you should keep in mind:

- If you want to multiply a Vector3D by a scalar, you must put the scalar at the right of the vector in the multiplication.
- If you want to multiply two Vector3D or two RGBColor element by element, you have available the *multiply()* method in both classes
- You must use the Lambert BRDF class.

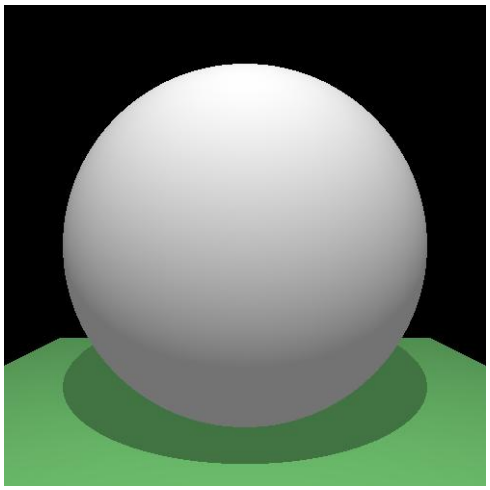

Figure 5. Expected result for the sphere scene when using the Phong integrator.

Validation. To verify the correctness of your implementation, you can compare your results to those of Figures 1 and 5.