2-gVirtualXRay_vs_Gate-monochromatic-80keV

March 2, 2022

[1]:
```python
from IPython.display import display
from IPython.display import Image
from utils import * # Code shared across more than one notebook
```

Main contributors: F. P. Vidal and J. M. Létang

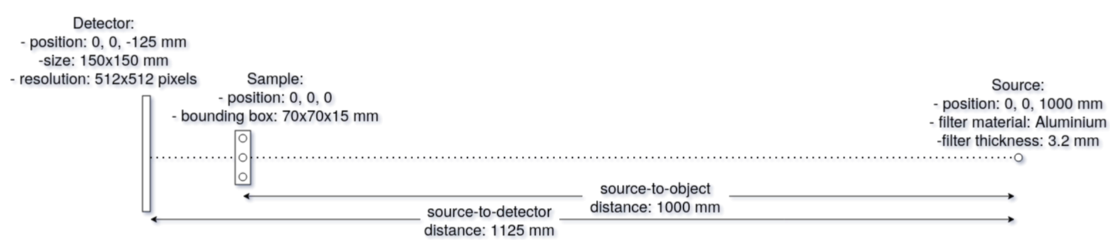
Purpose: In this notebook, we aim to demonstrate that gVirtualXRay is able to generate analytic simulations on GPU comparable to images generated with the state-of-the-art Monte Caro simulation packages. We use here a monochromatic source of 80 keV.

Material and Methods: We simulate an image with gVirtualXRay and compare it with a ground truth image. For this purpose, we use Gate, a wrapper for CERN's state-of-the-art Monte Caro simulation tool: Geant4. The number of tracked particles is 1e10.

In our simulation the source-to-object distance (SOD) is 1000mm, and the source-to-detector distance (SDD) is 1125mm. The beam spectrum is monochromatic, with an enegy of 80 keV.

[2]:
```python
Image(filename="doc/setup.png")
```

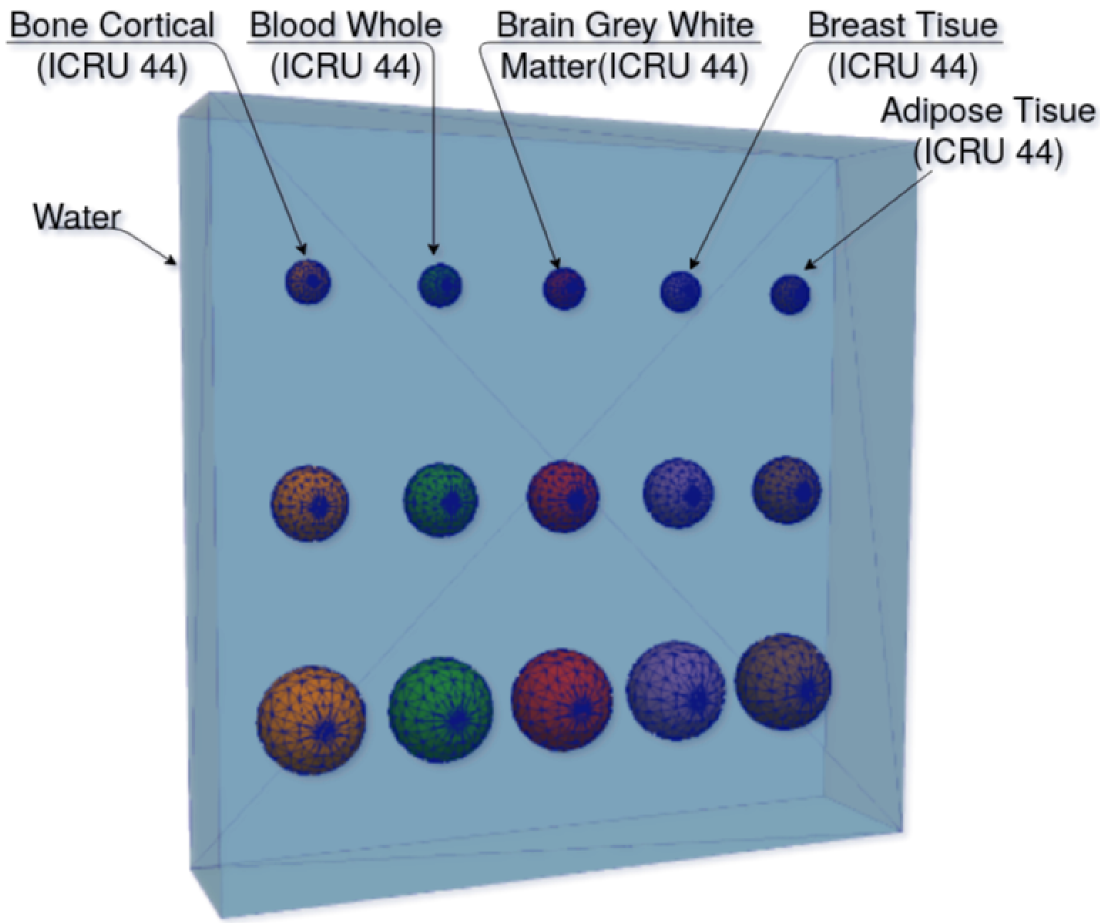
[2]:



The sample is made of a 70x70x15mm box of water, in which 5 columns of 3 spheres of different radii (2, 3.5, and 5mm) have been inserted. A given material is associated to the spheres of each column (bone (cortical), blood (whole), brain (grey/white matter), breast tissue, and adipose tissue). The columns are ordered in decreasing density. We use the definitions of tissue substitutes provided in the ICRU Report 44 by the International Commission on Radiation Units and Measurements. The material composition is available at `https://physics.nist.gov/PhysRefData/XrayMassCoef/tab2.html`.

[3]:
```python
Image(filename="doc/sample.png", width=400)
```

[3]:



Results: The calculations were performed on the following platform:

[4]:
```
printSystemInfo()
```

```
OS:
        Linux 5.3.18-150300.59.49-default
        x86_64

CPU:
        AMD Ryzen 7 3800XT 8-Core Processor

RAM:
        63 GB
GPU:
        Name: GeForce RTX 2080 Ti
        Drivers: 455.45.01
        Video memory: 11 GB
```

The Monte Carlo simulation needed 2.65e6 HS06 seconds to complete. It is equivalent to **1.15E+08** ms (i.e. ~1.3 day) on the system used. Only 7 ± 1 ms was needed with the GPU used.

The mean absolute percentage error (MAPE), also known as mean absolute percentage deviation (MAPD), between the two simulated images is **MAPE 0.45%**. The **zero-mean normalised cross-correlation is 99.87%**. The **Structural Similarity Index (SSIM) is 0.87**. As MAPE is low (close to 0), SSIM is 0.87, and ZNCC is high (close to 100%), we can conclude that this X-ray image simulated with gVirtualXRay on GPU in milliseconds is comparable to the same Monte Carlo simulation that ran for days.

1 Import packages

[5]:
```
%matplotlib inline

import os # Locate files

import math
import numpy as np # Who does not use Numpy?
import pandas as pd # Load/Write CSV files

import matplotlib

from matplotlib.cm import get_cmap
import matplotlib.pyplot as plt # Plotting
from matplotlib.colors import LogNorm # Look up table
from matplotlib.colors import PowerNorm # Look up table
import matplotlib.colors as mcolors

font = {'family' : 'serif',
         #'weight' : 'bold',
          'size'   : 22
        }
matplotlib.rc('font', **font)
# matplotlib.rc('text', usetex=True)

from scipy.stats import pearsonr # Compute the correlatio coefficient

from skimage.util import compare_images # Checkboard comparison between two␣
  ↪images
from skimage.metrics import structural_similarity as ssim
from sklearn.metrics import mean_absolute_percentage_error as mape
from skimage.metrics import structural_similarity as ssim

from tifffile import imread, imwrite # Load/Write TIFF files

import datetime # For the runtime
```

```python
import viewscad # Use OpenSCAD to create STL files

import gvxrPython3 as gvxr # Simulate X-ray images

import json2gvxr # Set gVirtualXRay and the simulation up
```

```
SimpleGVXR 1.0.1 (2022-02-22T14:00:25) [Compiler: GNU g++] on Linux
gVirtualXRay core library (gvxr) 1.1.5 (2022-02-22T14:00:25) [Compiler: GNU g++]
on Linux
```

2 Reference image

We first load the reference image that has been simulated using Gate wrapper for CERN's Geant4. Here we ignore scattering.

[6]:
```python
Image = imread("Gate_data/monoE_flat.tif") # Already corrected
Full_field = np.ones(Image.shape) # Perfect full field image
Dark_field = np.zeros(Full_field.shape) # Perfect dark field image
```

Projections are then corrected to account for variations in beam homogeneity and in the pixel-to-pixel sensitivity of the detector. This is the projection with flat-field correction (**Proj**):

$$\mathbf{Proj} = \frac{I - D}{F - D} \tag{1}$$

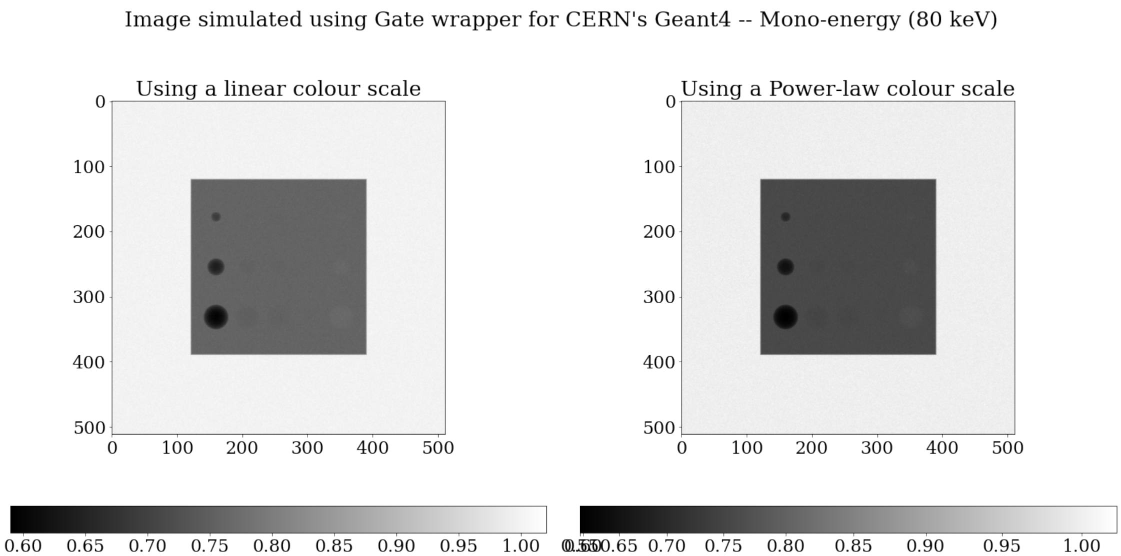
where F (full fields) and D (dark fields) are projection images without sample and acquired with and without the X-ray beam turned on respectively.

We now apply the flat-field correction to `Image`.

[7]:
```python
gate_image = (Image - Dark_field) / (Full_field - Dark_field)
# gate_image = Image / np.mean(Full_field)
```

We plot the image using a linear look-up table and a power-law normalisation.

[8]:
```python
displayLinearPowerScales(gate_image,
                         "Image simulated using Gate wrapper for CERN's Geant4␣
 ↪-- Mono-energy (80 keV)",
                         "plots/reference_from_Gate-monoE-80keV")
```



3 Setting up gVirtualXRay

Before simulating an X-ray image using gVirtualXRay, we must create an OpenGL context.

[9]:
```
json2gvxr.initGVXR("notebook-2.json", "EGL")
```

```
Create an OpenGL context: 800x450

0

Wed Mar  2 12:34:18 2022 ---- Create window gvxrStatus: Create window
Wed Mar  2 12:34:19 2022 ---- EGL version: Wed Mar  2 12:34:19 2022 ---- OpenGL
version supported by this platform OpenGL renderer:   GeForce RTX 2080
Ti/PCIe/SSE2
OpenGL version:    4.5.0 NVIDIA 455.45.01
OpenGL vender:     NVIDIA Corporation
Wed Mar  2 12:34:19 2022 ---- Use OpenGL 4.5.0 0 500 500
0 0 800 450

1.5
4.5.0 NVIDIA 455.45.01
```

3.1 X-ray source

We create an X-ray source. It is a point source.

[10]:
```
json2gvxr.initSourceGeometry()
```

```
Set up the beam
	Source position: [0.0, 0.0, 1000.0, 'mm']
	Source shape: PointSource
```

3.2 Spectrum

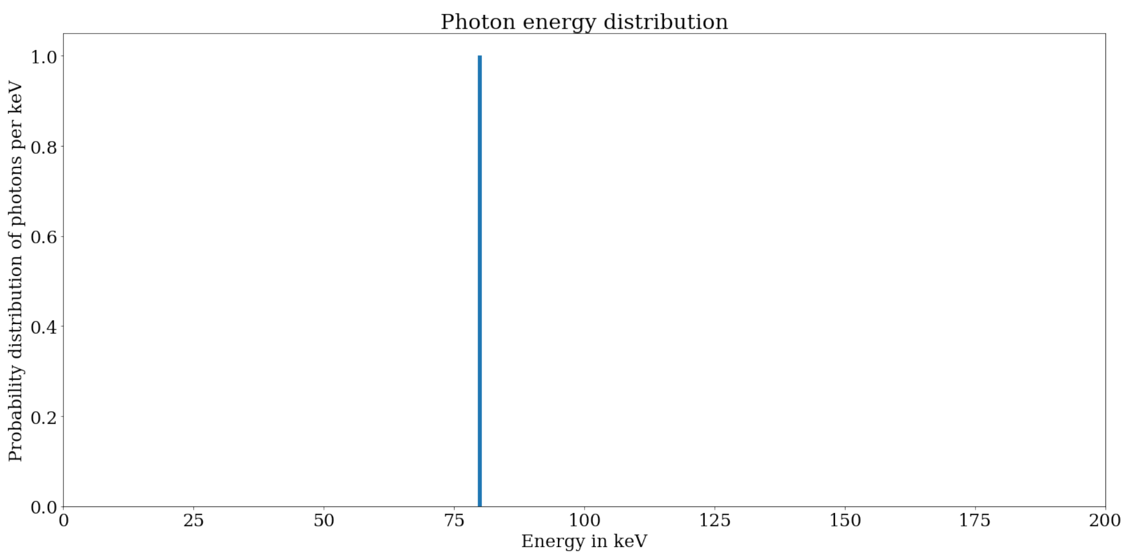
The spectrum is monochromatic, with an energy of 80 keV.

[11]:
```python
spectrum, unit, k, f = json2gvxr.initSpectrum(verbose=1)
energy_set = sorted(spectrum.keys())

count_set = []

for energy in energy_set:
    count_set.append(spectrum[energy])
```

```
1.0 photon of 80 keV
```

Plot the spectrum

[12]:
```python
plotSpectrum(k, f, 'plots/spectrum-monoE-80keV')
```



3.3 Detector

Create a digital detector

[13]:
```python
json2gvxr.initDetector()
```

```
Set up the detector
        Detector position: [0.0, 0.0, -125.0, 'mm']
        Detector up vector: [0, 1, 0]
        Detector number of pixels: [512, 512]
        Pixel spacing: [0.29296875, 0.29296875, 'mm']
```

3.4 Create and load the test object

We now create CAD models using OpenSCAD and extract the corresponding STL files.

[14]:
```
openscad_make_spheres_str = """

module make_column_of(sphere_radius, height, count)
{
    step = height / (count - 1);
    for (a = [0 : count - 1]) {
        offset = -height / 2 + step * a ;
        translate([0, offset, 0])
            sphere(sphere_radius[a], $fn=25);
    }
}

module make_row_of(radius, count, id)
{
    step = radius / (count - 1);
    for (a = [0 : count - 1]) {
        if (id == -1 || id == a) {
            offset = -radius / 2 + step * a ;
            translate([offset, 0, 0])
                children();
        }
    }
}

module make_spheres(sphere_radius, ring_radius, ring_count, column_height,␣
 ↪column_count, id = -1)
{
    make_row_of(radius = ring_radius, count = ring_count, id = id)
        make_column_of(sphere_radius, height = column_height, count =␣
 ↪column_count);
}
"""
```

The matrix

[15]:
```
openscad_matrix_str = """

color("red")
    difference() {
        scale([70, 70, 15])
            cube(1, center = true);
        make_spheres([2, 3.5, 5], 50, 5, 40, 3, -1);
}

"""
```

[16]:
```
fname = 'CAD_models/matrix.stl'
if not os.path.isfile(fname):

    r = viewscad.Renderer()
    r.render(openscad_matrix_str + openscad_make_spheres_str, outfile=fname)
```

[17]:
```
openscad_cube_str = """

color("red")
    scale([70, 70, 15])
        cube(1, center = true);

"""
```

[18]:
```
fname = 'CAD_models/cube.stl'
if not os.path.isfile(fname):

    r = viewscad.Renderer()
    r.render(openscad_cube_str, outfile='gvxr/input/cube.stl')
```

The spheres

[19]:
```
openscad_col_str_set = []

for i in range(5):
    openscad_col_str_set.append("""
    color("blue")
        make_spheres([2, 3.5, 5], 50, 5, 40, 3, """ + str(i) + ");")

    fname = 'CAD_models/col_' + str(i) + '.stl'
    if not os.path.isfile(fname):

        r = viewscad.Renderer()
        r.render(openscad_col_str_set[-1] + openscad_make_spheres_str,␣
 ↪outfile=fname)
```

Load the samples. `verbose=2` is used to print the material database for Gate. To disable it, use verbose=0 or verbose=1.

[20]:
```python
json2gvxr.initSamples(verbose=1)
```

```
Load the 3D data

        Load Bone_Cortical_ICRU_44 in CAD_models/col_0.stl using mm
        Load Blood_Whole_ICRU_44 in CAD_models/col_1.stl using mm
        Load Brain_Grey_White_Matter_ICRU_44 in CAD_models/col_2.stl using mm
        Load Breast_Tissue_ICRU_44 in CAD_models/col_3.stl using mm
        Load Adipose_Tissue_ICRU_44 in CAD_models/col_4.stl using mm
        Load H2O in CAD_models/cube.stl using mm

CAD_models/col_0.stl    nb_faces:       1938    nb_vertices:    5814
bounding_box (in cm):   (-2.99606, -2.19961, -0.496354) (-2, 2.49901, 0.496354)
CAD_models/col_1.stl    nb_faces:       1938    nb_vertices:    5814
bounding_box (in cm):   (-1.74606, -2.19961, -0.496354) (-0.75, 2.49901,
0.496354)
CAD_models/col_2.stl    nb_faces:       1938    nb_vertices:    5814
bounding_box (in cm):   (-0.496057, -2.19961, -0.496354)        (0.5, 2.49901,
0.496354)
CAD_models/col_3.stl    nb_faces:       1938    nb_vertices:    5814
bounding_box (in cm):   (0.753943, -2.19961, -0.496354) (1.75, 2.49901,
0.496354)
CAD_models/col_4.stl    nb_faces:       1938    nb_vertices:    5814
bounding_box (in cm):   (2.00394, -2.19961, -0.496354)  (3, 2.49901, 0.496354)
CAD_models/cube.stl     nb_faces:       12      nb_vertices:    36
bounding_box (in cm):   (-3.5, -3.5, -0.75)     (3.5, 3.5, 0.75)
```

4 Run the simulation

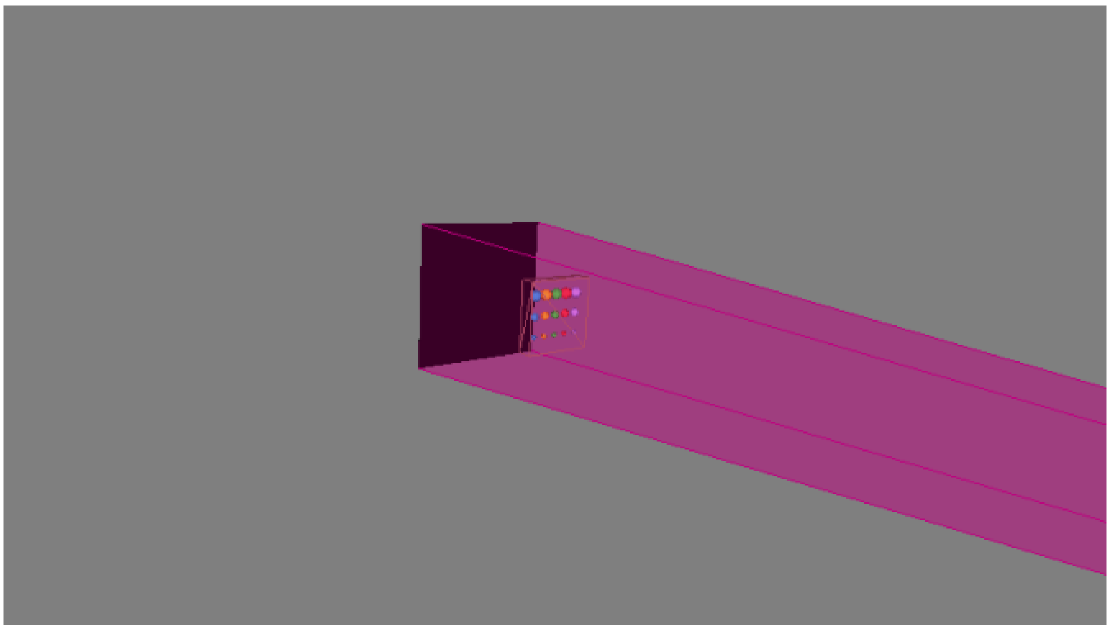
Update the 3D visualisation and take a screenshot

[21]:
```python
gvxr.displayScene()

gvxr.useLighing()
gvxr.useWireframe()
gvxr.setZoom(719.6787109375)
gvxr.setSceneRotationMatrix([0.7624880075454712, 0.09040657430887222, -0.
↪6406543850898743, 0.0,
                             0.05501500517129898, 0.9775413870811462, 0.
↪20342488586902618, 0.0,
                             0.6446591019630432, -0.190354123711586, 0.
↪7403913140296936, 0.0,
                             0.0, 0.0, 0.0, 1.0])

gvxr.displayScene()
```

[22]:
```
screenshot = gvxr.takeScreenshot()
```

[23]:
```
plt.figure(figsize= (10,10))
plt.title("Screenshot")
plt.imshow(screenshot)
plt.axis('off')

plt.tight_layout()

plt.savefig('plots/screenshot-beam-off-monoE-80keV.pdf')
plt.savefig('plots/screenshot-beam-off-monoE-80keV.png')
```

Screenshot



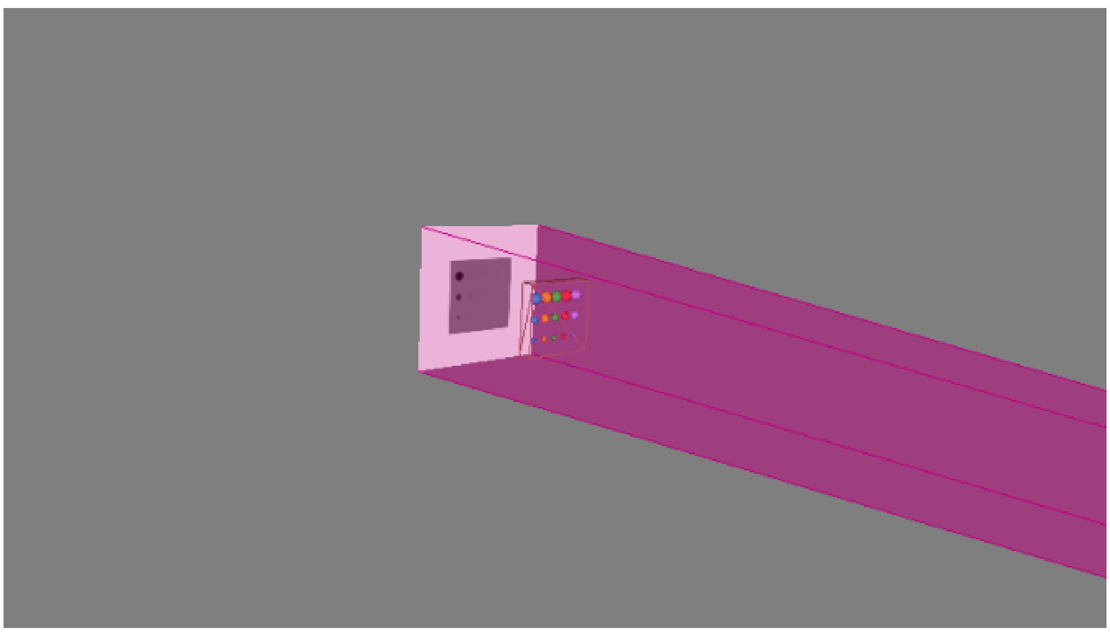
Compute an X-ray image 100 times (to gather performance statistics)

[24]:
```
# gvxr.enableArtefactFilteringOnCPU()
gvxr.enableArtefactFilteringOnGPU()
# gvxr.disableArtefactFiltering() # Spere inserts are missing with GPU␣
↪integration when a outer surface is used for the matrix


runtimes = []

for i in range(100):
    start_time = datetime.datetime.now()
    gvxr.computeXRayImage()
```

```
    end_time = datetime.datetime.now()
    delta_time = end_time - start_time
    runtimes.append(delta_time.total_seconds() * 1000)

gvxr.displayScene()
```

[25]:
```
screenshot = gvxr.takeScreenshot()

plt.figure(figsize= (10,10))
plt.title("Screenshot")
plt.imshow(screenshot)
plt.axis('off')

plt.tight_layout()

plt.savefig('plots/screenshot-beam-on-monoE-80keV.pdf')
plt.savefig('plots/screenshot-beam-on-monoE-80keV.png')
```

Screenshot



Save an X-ray image

[26]:
```
# Compute the L-buffers on the GPU and integrate on the GPU
x_ray_image_integration_GPU = np.array(gvxr.computeXRayImage())
imwrite('gVirtualXRay_output_data/projection_raw_integration_GPU_monoE-80keV.
 ↪tif', x_ray_image_integration_GPU.astype(np.single))
```

Flat-field correction

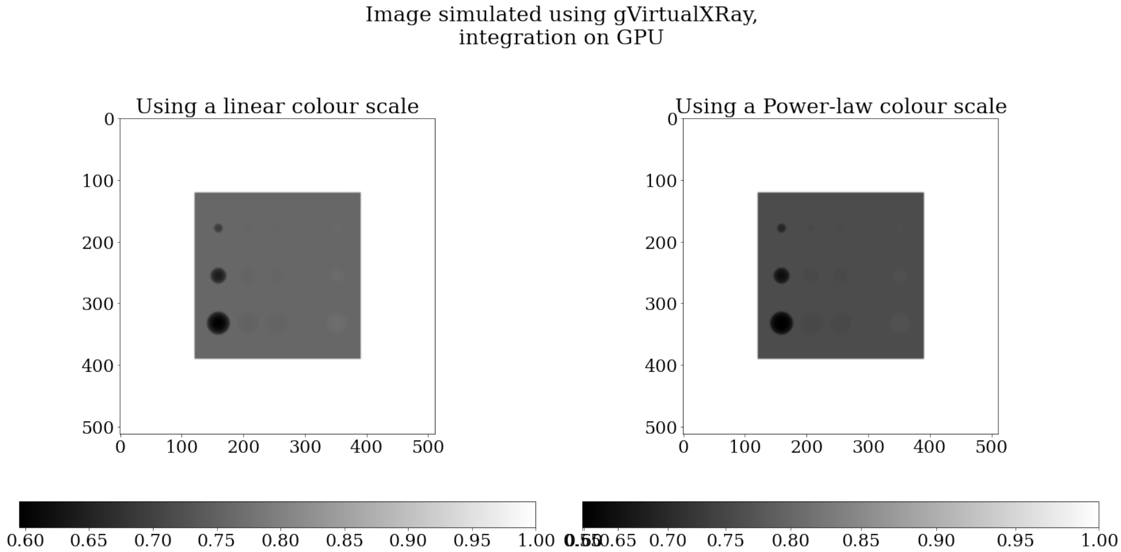
[27]:
```
total_energy_in_keV = 0.0
for energy, count in zip(energy_set, count_set):
    total_energy_in_keV += energy * count

total_energy_in_MeV = gvxr.getTotalEnergyWithDetectorResponse()
```

[28]:
```
white = np.ones(x_ray_image_integration_GPU.shape) * total_energy_in_MeV
dark = np.zeros(x_ray_image_integration_GPU.shape)

x_ray_image_integration_GPU = (x_ray_image_integration_GPU - dark) / (white -␣
↪dark)
```

Save the corresponding image

[29]:
```
imwrite('gVirtualXRay_output_data/
↪projection_corrected_integration_GPU_monoE-80keV.tif',␣
↪x_ray_image_integration_GPU.astype(np.single))
```

[30]:
```
plt.figure(figsize= (20,10))

plt.suptitle("Image simulated using gVirtualXRay,\nintegration on GPU", y=1.02)

plt.subplot(121)
plt.imshow(x_ray_image_integration_GPU, cmap="gray")
plt.colorbar(orientation='horizontal')
plt.title("Using a linear colour scale")

plt.subplot(122)
plt.imshow(x_ray_image_integration_GPU, norm=PowerNorm(gamma=1./0.75),␣
↪cmap="gray")
plt.colorbar(orientation='horizontal')
plt.title("Using a Power-law colour scale")

plt.tight_layout()

plt.savefig('plots/x_ray_image_integration_GPU-monoE-80keV.pdf')
plt.savefig('plots/x_ray_image_integration_GPU-monoE-80keV.png')
```



5 Comparison the analytic simulation with the Monte Carlo simulation

5.1 Quantitative validation

Compute image metrics between the two simulated images:

1. mean absolute percentage error (MAPE), also known as mean absolute percentage deviation (MAPD),
2. zero-mean normalised cross-correlation (ZNCC), and
3. Structural Similarity Index (SSIM).

We use these three metrics as one is a disimilarity measurement (MAPE), two are similarity measurement (ZNCC & SSIM). MAPE and ZNCC can be expressed as a percentage, which eases the interpretation of the numerical values. SSIM is a number between 0 and 1. A good value of MAPE s 0%; of ZNCC 100%, and SSIM 1.

[31]:
```
MAPE_integration_GPU = mape(gate_image, x_ray_image_integration_GPU)
ZNCC_integration_GPU = np.mean((gate_image - gate_image.mean()) / gate_image.
 ↪std() * (x_ray_image_integration_GPU - x_ray_image_integration_GPU.mean()) /␣
 ↪x_ray_image_integration_GPU.std())
SSIM_integration_GPU = ssim(gate_image, x_ray_image_integration_GPU,␣
 ↪data_range=gate_image.max() - gate_image.min())

print("MAPE_integration_GPU:", "{0:0.2f}".format(100 * MAPE_integration_GPU) +␣
 ↪"%")
print("ZNCC_integration_GPU:", "{0:0.2f}".format(100 * ZNCC_integration_GPU) +␣
 ↪"%")
```

```
print("SSIM_integration_GPU:", "{0:0.2f}".format(SSIM_integration_GPU))
```

```
MAPE_integration_GPU: 0.45%
ZNCC_integration_GPU: 99.87%
SSIM_integration_GPU: 0.87
```

Get the total number of triangles

[32]:
```
number_of_triangles = 0

for mesh in json2gvxr.params["Samples"]:
    label = mesh["Label"]
    number_of_triangles += gvxr.getNumberOfPrimitives(label)
```

[33]:
```
runtime_avg = round(np.mean(runtimes))
runtime_std = round(np.std(runtimes))
```

Print a row of the table for the paper

[34]:
```
print("Sphere inserts -- mono energy (80 keV) & Gate & " +
      "{0:0.2f}".format(100 * MAPE_integration_GPU) + "\\%    &    " +
      "{0:0.2f}".format(100 * ZNCC_integration_GPU) + "\\%    &    " +
      "{0:0.2f}".format(SSIM_integration_GPU) + "    &    $" +
      str(json2gvxr.params["Detector"]["NumberOfPixels"][0]) + " \\pm " +␣
 ↪str(json2gvxr.params["Detector"]["NumberOfPixels"][1]) + "$    &    " +
      str(number_of_triangles) + "    &    " +
      "1.15E+08    &    " +
      "$" + str(runtime_avg) + " \\pm " + str(runtime_std) + "$ \\\\")
```

```
Sphere inserts -- mono energy (80 keV) & Gate & 0.45\%    &    99.87\%    &
0.87    &    $512 \pm 512$    &    9702    &    1.15E+08    &    $7 \pm 1$ \\
```

In both cases, MAPE is very small (less than 1%) and ZNCC is very high (more than 99%). We can conclude that the two images are similar. The main difference lie in the Poisson noise affecting the Monte Carlo simulation.

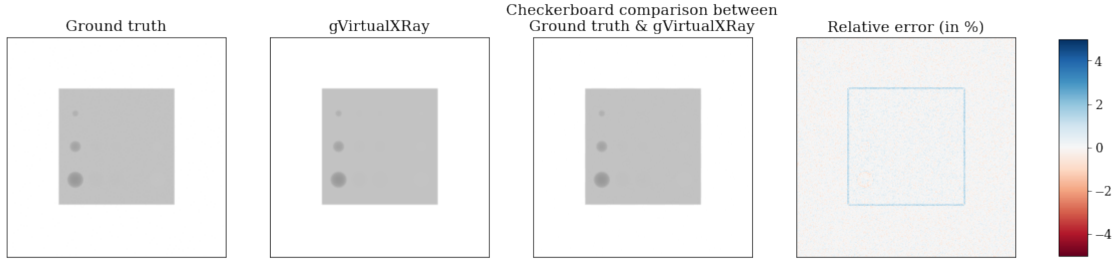
5.2 Qualitative validation

Checkboard comparison

[35]:
```
font = {'size'   : 12.5
       }
matplotlib.rc('font', **font)
```

[36]:
```
fullCompareImages(gate_image=gate_image,
                  gvxr_image=x_ray_image_integration_GPU,
                  title="gVirtualXRay",
                  fname="plots/full_comparison_integration_GPU-monoE-80keV",
```

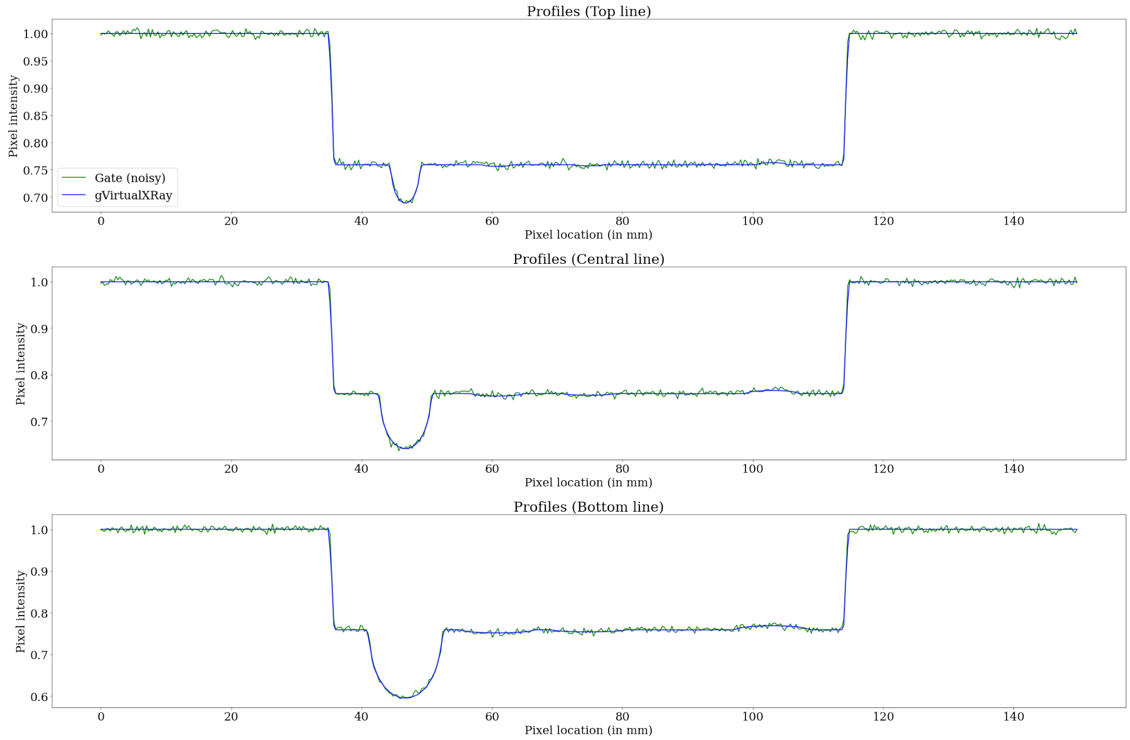
```
                        log=False,
                        vmin=0.0,
                        vmax=1,
                        avoid_div_0 = False)
```



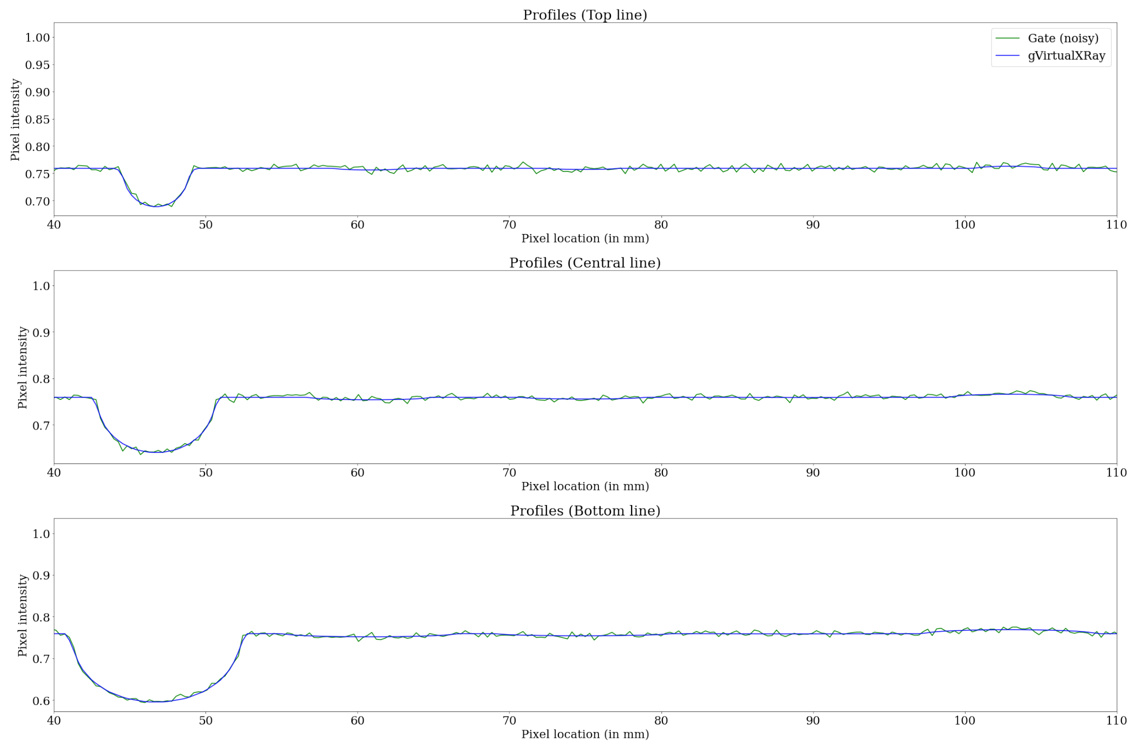
Plot the profiles

[37]:
```
font = {'size'   : 22
       }
matplotlib.rc('font', **font)
```

[38]:
```
plotTwoProfiles(json2gvxr, gate_image, x_ray_image_integration_GPU, "plots/
↪profiles-checkerboard-monoE-80keV")
```



[39]:
```
spacing = json2gvxr.params["Detector"]["Size"][0] / json2gvxr.
 ↪params["Detector"]["NumberOfPixels"][0]
min_limit = round(40)
max_limit = round(512 * spacing - 40)
plotTwoProfiles(json2gvxr, gate_image, x_ray_image_integration_GPU, "plots/
 ↪profiles-zoom-checkerboard-monoE-80keV", [min_limit, max_limit])
```



6 All done

Destroy the window

[40]:
```
gvxr.destroyAllWindows()
```

0(0x5568afb22a70)