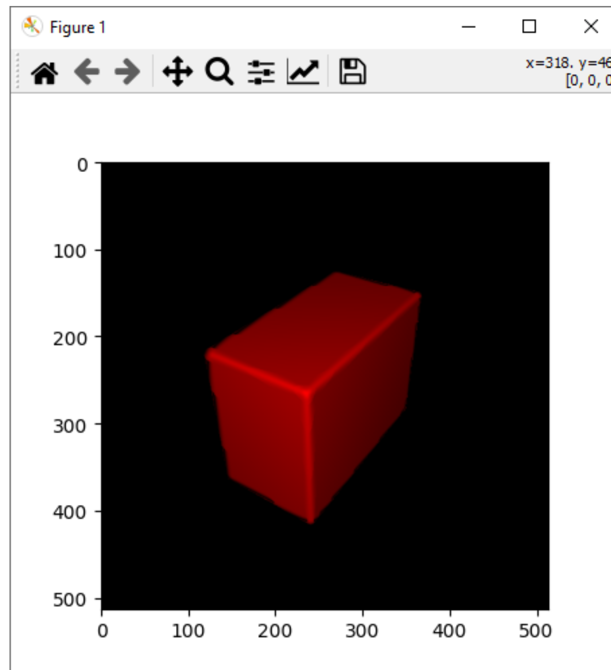
A simple GPU-powered raymarcher in Python – Part II

Motivated by the outcomes in **Part I of this article**, I liked to extend the Numba-based raymarcher by camera movements via mouse events, perspective projection and shading. In my example (see [1] for the complete code), a red cuboid is raymarched in real-time.



Raymarched cuboid with Numba in Matplotlib

Dynamic images and interactivity in Python

Firstly, I searched for a windowing system where the raymarched images could be pasted continuously, what is required to support camera movements or animations. PyQt would be one candidate but it's rather a large framework for creating applications with a graphical user interface. A more lightweight solution is Matplotlib, which I chose for my example. In a plot of this library, it is possible to add array-like images by *im = plt.imshow(…)* and update the data in a custom rendering loop by *im.set_data(…)*. In each iteration of the rendering loop, the data is changed by calling the Numba kernel. The methods *draw()* and *flushEvents()* of *ax.figure.canvas* create new frames, whereat *ax = plt.gca()* is the main axis of the plot. As a precondition, you have to turn on the interactive mode by *plt.ion()*. You can add listeners to mouse events like pressing and releasing buttons, moving or scrolling by *ax.figure.canvas.mpl_connect(…)*.

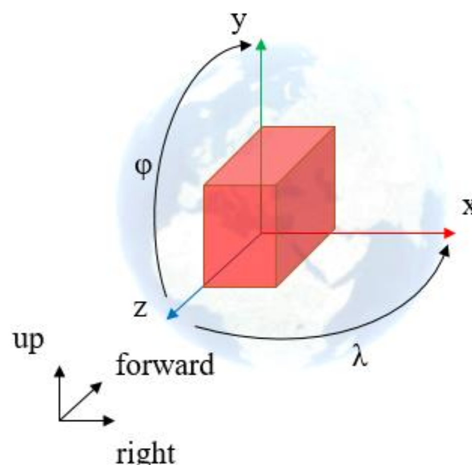
```
plt.ion()
ax = plt.gca()

canvas = ax.figure.canvas

canvas.mpl_connect("button_press_event", on_press)
canvas.mpl_connect("button_release_event", on_release)
canvas.mpl_connect("motion_notify_event", on_move)
canvas.mpl_connect("scroll_event", on_scroll)
canvas.mpl_connect("close_event", on_close)

result = np.zeros((image_size, image_size, 3), dtype=np.uint16)
im = plt.imshow(result)
```

```
# rendering loop
while(True):
    result = np.zeros((image_size, image_size, 3), dtype=np.uint16)
    my_kernel[image_size, image_size](result, camera_position, camera_forward,
              camera_up, camera_right, camera_distance)
    im.set_data(result)

    canvas.draw()
    canvas.flush_events()
```

Camera movements via mouse events

In my example, I mapped mouse deviations in x and y direction to geographical coordinates λ and φ (= longitude, latitude) of a virtual spherical earth, so that one can look at the raymarched shape from all sides from a certain distance r. I set the initial position of the camera in Cartesian coordinates to (0, 0, r) in a right-handed coordinate system. The initial forward vector (aka look vector) of the camera is along the negative z-axis, the up vector corresponds to the positive y-axis and the right vector to the positive x-axis. The initial geographical coordinates are (λ, φ) = (0, 0). When you move along the equator to (90°, 0), you will be at position (r, 0, 0), up will still be the positive y-axis and right the negative z-axis. When you move to the north pole to (90°, 90°), you will be at position (0, r, 0), up will be aligned to the negative x-axis and right to the negative z-axis. While the forward vector of the camera is its inverted position, the up and right camera vector can be obtained by these instructions. Note that you have to restrict the latitude to]-90°, 90°[to prevent overflows at the poles. For rotating the position vector, I used the *Rotation.apply(…)* function from SciPy as described in this kite article. r can be adjusted by scrolling the mouse wheel.

```
camera_coords = np.array([0.0, 0.0])
target_point = np.array([0.0, 0.0, 0.0])
camera_distance = 128
lon = camera_coords[0]
lat = camera_coords[1]
camera_position = [0, 0, camera_distance]
rotation_x = Rotation.from_rotvec([lat, 0, 0])
camera_position = rotation_x.apply(camera_position)
rotation_y = Rotation.from_rotvec([0, lon, 0])
camera_position = rotation_y.apply(camera_position)
camera_forward = normalize(target_point - camera_position)
camera_right = normalize(np.cross(camera_forward, np.array([0, 1, 0])))
camera_up = np.cross(camera_right, camera_forward)
```



Cartesian coordinate system and camera rotation using geographical coordinates

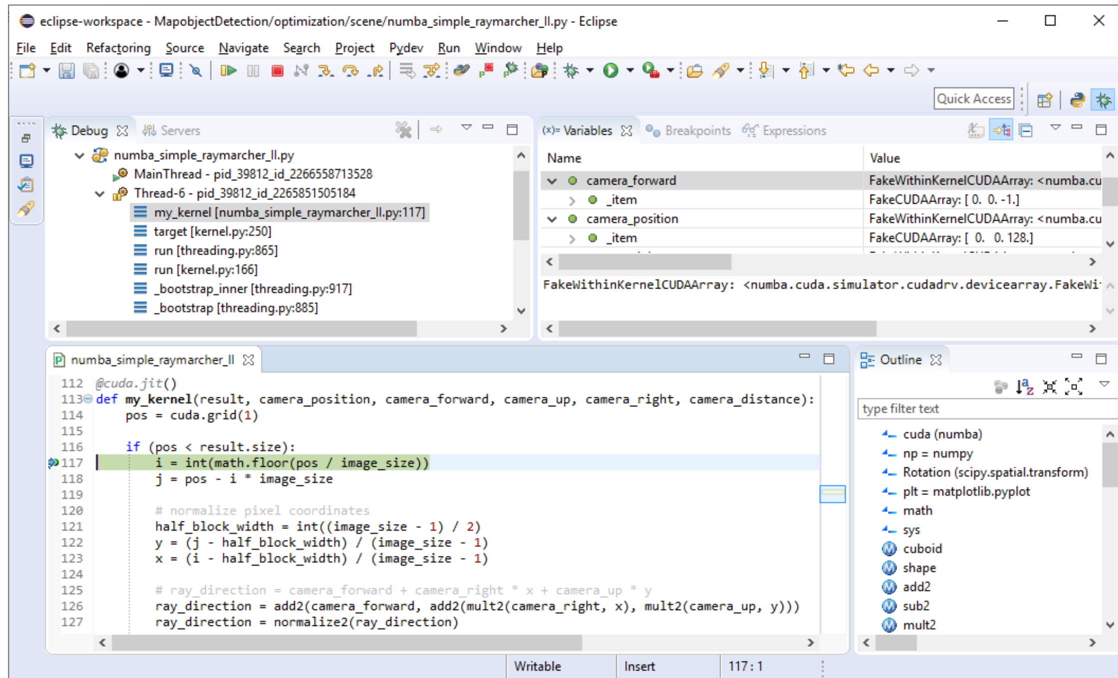
Perspective projection and shading

The camera vectors are next passed to the kernel function so that rays can be cast from the camera position. The ray direction is calculated as *forward + right * x + up * y,* where x and y are pixel coordinates normalized to [-0.5, 0.5]. The ray position is *camera position + stepped distance * ray direction*, where the stepped distance is increased each loop according to the distance to the raymarched object. In my example, a cuboid is raymarched by using this signed distance function. When the stepped distance is too large, then the loop is quit. I used 2r as a threshold in my example. When the distance is below a certain epsilon, then the object is hit. By then inspecting the neighborhood of the current position of the ray, the surface normal can be estimated according this formula. The light intensity can then be interpreted as *|dot(surface normal, light direction)|*, whereat both vectors are normalized. In my example, I have taken the ray direction as light direction, which means that the light source is the camera. In the initial setup, the normal of the visible surface is (0, 0, 1) and the ray direction of the center pixel is (0, 0, -1), what leads to the full light intensity. When you would look at the same surface from another angle, the z value of the ray direction would decrease, resulting in a lower intensity.

```
pos = cuda.grid(1)
image_size = 513

i = int(math.floor(pos / image_size))
j = pos - i * image_size

# normalize pixel coordinates
half_block_width = int((image_size - 1) / 2)
y = (j - half_block_width) / (image_size - 1)
x = (i - half_block_width) / (image_size - 1)

# ray_direction = camera_forward + camera_right * x + camera_up * y
ray_direction = add2(camera_forward, add2(mult2(camera_right, x), mult2(camera_up, y)))
ray_direction = normalize2(ray_direction)

stepped_distance = 0.0

while(True):
    # ray_point = camera_position + ray_direction * stepped_distance
    ray_point = add2(camera_position, mult2(ray_direction, stepped_distance))

    distance = shape(ray_point)

    if (distance < 1):
        surface_normal = estimate_normal(ray_point)
        light_intensity = abs(dot2(surface_normal, ray_direction))

        result[j][i][0] = light_intensity * 255
        result[j][i][1] = 0
        result[j][i][2] = 0
        break

    stepped_distance += distance

    if (stepped_distance > camera_distance*2):
        break
```

Numba strengths and limitations

While implementing the upper vector calculations, I see the possibility to debug the kernel function as a strength of Numba. If you add the environment variable *NUMBA_ENABLE_CUDASIM=1*, all threads will run on the CPU, what allows to print values or to set break points. Since too many threads slow down the debugger, it is advisable to reduce the image size beforehand.



Debugging Numba kernels in Eclipse with PyDev

Numba supports a broad range of Python math functions in CUDA kernels, however only a limited part of Numpy features. As suggested in this stackoverflow answer, Numpy is the preferred choice for handling vector operations. For example, you cannot use the add operator on two arrays, thus you have to implement the element-wise addition on your own. Similar applies to scalar multiplication, dot products and normalization. Another limitation is that you can declare arrays with *cuda.local.array(shape=3, dtype=numba.float32)*, but you cannot return them in helper functions. Therefore, I used tuples instead of arrays in helper functions and updated their elements. It would be nice if more linear algebra functionality could be natively supported in Numba, as it is for instance the case with WebGL. This may be the case when overloading of Python operators will be implemented.

Overall, I was happy that I could achieve my targeted result. Maybe I will further improve it, but up to now, this is the last part of my article. I hope that my example is helpful for your research. Feel free to extend or simplify it!

Gist

[1] Simple raymarcher with Python and Numba (Part II)

Raimund Schnürer, 14.04.2021 (updated 17.09.2021)

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.