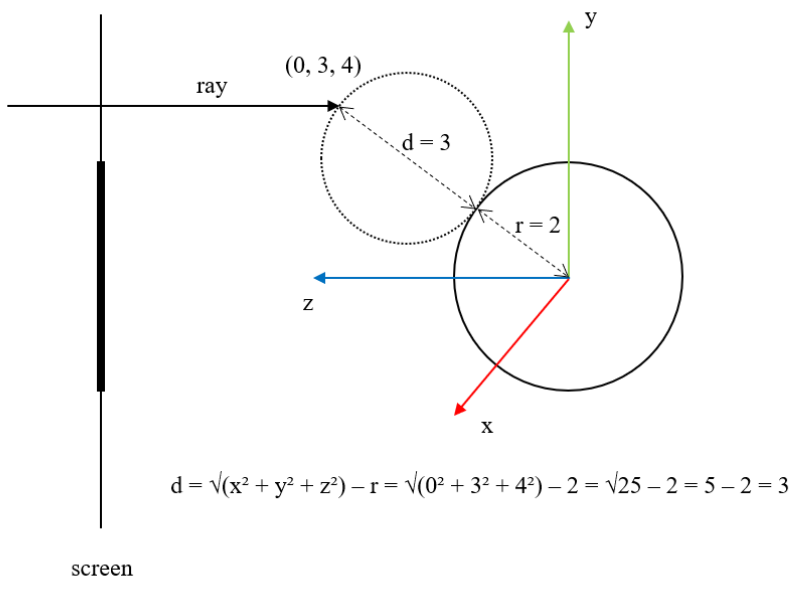
A simple GPU-powered raymarcher in Python – Part I

Sphere tracing is a special ray marching/ray tracing algorithm for rendering implicit surfaces [Hart 1996]. The algorithm enables to create 3D scenes in real-time by making use of the parallelization capabilities of the GPU. On your quest seeking for more information, you will certainly come across several resources provided by Inigo Quilez, like one of his informative blog posts, 3D signed distance functions (SDFs) on his website, or some of his examples on Shadertoy. But there are lots other learning materials like interactive tutorials, other explanatory articles, or illustrative videos.

Recently, implicit surfaces got more and more attention in the machine learning (ML) community to approximate 3D shapes. Stunning results have been achieved in morphing of 3D objects [Park et al. 2019] or reconstructing 3D models of humans from 2D images [Saito et al. 2019].

PIFu: Pixel-Aligned Implicit Function for High-Resolution Clothed Human Digitization [Saito et al. 2019]

I also started research in this domain and I was looking for a convenient method to visualize the implicit surfaces included in my training or test dataset. Park et al. have used C++ and OpenGL to render the images, for instance, whereas Saito et al. relied on PyOpenGL. As major ML frameworks like TensorFlow or PyTorch are based on Python, I preferred the second approach because it sticks to the same programming language. But the code is quite lengthy, that is why I wondered whether this can be achieved more easily. On a quick search on GitHub, I found only CPU-based implementations of raymarchers in Python, so I tried to be the architect of my own fortune.


y
ray
(0, 3, 4)
d = 3
r = 2
z
x
$d = \sqrt{(x^2 + y^2 + z^2)} - r = \sqrt{(0^2 + 3^2 + 4^2)} - 2 = \sqrt{25} - 2 = 5 - 2 = 3$
screen
Sphere (dashed circle) tracing a sphere (solid circle) from side view

Not enough flow in TensorFlow

Since I’m familiar with TensorFlow, my first attempt was targeting the map_fn and vectorized_map functions, which can be applied element-wise to an array. map_fn turned out to be really slow, what has been also discussed in this issue. With vectorized_map, however, you can also use limited control structures (e.g. no native if-condition or for-loop) currently, so one has to rely on the TensorFlow functions (e.g. tf.cond, tf.while_loop). After some code shuffling, I came up with a working solution (see [1] for the complete code), where rendering a sphere on a 65x65px image took about 100ms on a GeForce GTX 1080. In its current state, the snippet is too slow for real-time rendering and coding was definitely no pleasure. Feel free to give me suggestions for further optimization. Otherwise let’s hope for future improvements of TensorFlow.

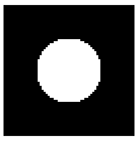
```
tf.function(input_signature=(
    tf.TensorSpec(shape=[None, 2], dtype=tf.int8),
    tf.TensorSpec(shape=[], dtype=tf.float32),
    tf.TensorSpec(shape=[], dtype=tf.float32)
))

def wrapper(indices, r, half_image_size):
    def raymarch(index_pair):
        x = tf.cast(index_pair[0], tf.float32)
        y = tf.cast(index_pair[1], tf.float32)
        z = half_image_size
        keep_looping = True

        def move(z):
            return tf.cond(tf.less(z, -half_image_size),
                           lambda: [False, 0.], lambda: [True, z])

        def trace(_, z):
            dist = sphere(x, y, z, r)

            return tf.cond(tf.less(dist, 1.), lambda: [False, 1.],
                   lambda: move(z - dist))

        [keep_looping, z] = tf.while_loop(lambda keep_looping, _:
                                          keep_looping, trace,
                                          [keep_looping, z])

        return tf.cast(z, tf.int8)

    return tf.vectorized_map(raymarch, indices)

image_size = 65
half_image_size = int((image_size - 1) / 2)
r = 15

indices = np.indices((image_size, image_size))
indices -= np.int((image_size - 1) / 2)
indices = np.moveaxis(indices, 0, -1)
indices = np.reshape(indices, [image_size*image_size, 2])
indices = tf.convert_to_tensor(indices, tf.int8)

result = wrapper(indices, r, half_image_size)
```

Numba to the rescue

Afterwards, I looked for alternative GPU-based Python libraries because I do not know PyTorch. I stumbled upon Numba which supports NVIDIA's CUDA and AMD's ROCm drivers to parallelize algorithms. Luckily, I once had read the book CUDA by Example, so I had at least some theoretical background. In Numba, you can simply decorate Python functions with *@cuda.jit* to declare them as kernels and instantiate them with a number of blocks and a number of threads per block. You can simply use the width and height for images for these numbers, since the rays are fired through each pixel. In the kernel function, you can access the current position by the variables *cuda.blockIdx.x* and *cuda.threadIdx.x*. You can pass parameters to the kernel function such as an array to store the result or some scene variables. Within the kernel function, you can use control structures and Python math functions out of the box. Only helper functions, for example for SDFs, need to be decorated with *@cuda.jit(device=True)*. Overall, the first call of the kernel function takes some milliseconds and the timings for the following calls are rounded to zero (see [2] for the complete code).

```
@cuda.jit
def my_kernel(result, r):
    i = cuda.blockIdx.x
    j = cuda.threadIdx.x

    x = i - half_image_size
    y = j - half_image_size
    z = half_image_size

    while(True):
        dist = sphere(x, y, z, r)

        if (dist < 1):
            result[j][i] = 1
            break

        z -= dist

        if (z < -half_image_size):
            break

image_size = 65
half_image_size = int((image_size - 1) / 2)
r = 15

result = np.zeros((image_size, image_size))

my_kernel[image_size, image_size](result, r)
```

In both of my examples, a white sphere is rendered on black background. For simplicity, rays are casted orthogonally to the z-axis and the sphere is not shaded.


The result: A simple raymarched sphere

If you are interested in learning more, I will describe in **Part II of this article** how to implement camera movements via mouse events, perspective projection and shading for the Numba-based raymarcher.

Gists

[1] Simple raymarcher with Python and TensorFlow
[2] Simple raymarcher with Python and Numba (Part I)

Raimund Schnürer, 14.04.2021