Coursework III:
Path Tracing

COMP0027 Team
Tobias Ritschel, Michael Fischer, Pradyumna (Preddy) Reddy, David Walton

November 28, 2022

We have shown you the framework for solving the coursework at `https://uclcg.github.io/uclcg/`.

You should start by extending the respective example there. The programming language is WebGL (`https://www.khronos.org/registry/webgl/specs/latest/1.0/`) and the OpenGL ES Shading Language (GLSL) `www.khronos.org/files/opengles_shading_language.pdf`. This should run in any browser, but we formerly experienced problems with Safari and thus recommend using a different browser such as Chrome. Do not write any answers in any other programming language, in paper, or pseudo code. Do not write code outside the `#define` blocks.

Remember to save your solution often enough to a `.uclcg` file. In the end, hand in that file via Moodle.

The total points for this exercise is **100**.

Please refer to Moodle for the due dates.

Introduction We have prepared a simple path tracing framework you would be asked to extend. The framework is very similar to the solution of the first coursework (ray-tracing), just that we removed cylinder intersections and point lights.

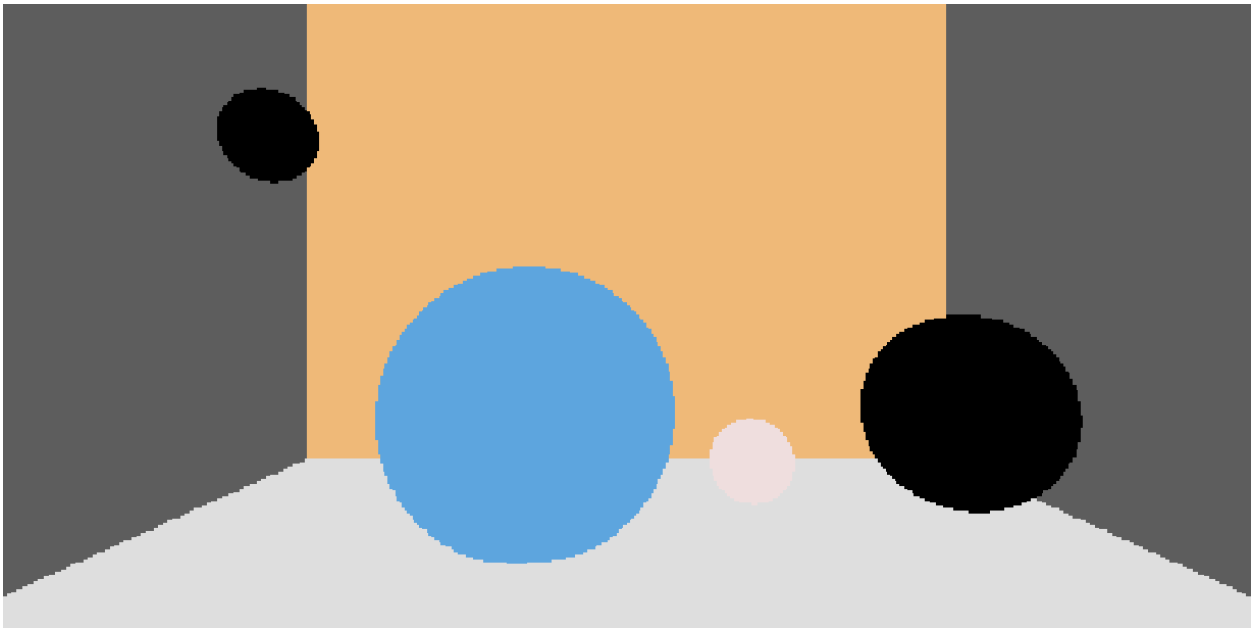
As in the previous coursework, we proceed in an artificially-low resolution for two reasons: Slow computers and in order to allow you to see individual pixels to notice subtle effects. The new revision of the framework allows to change the resolutions with a new button.

The path tracer is *progressive*: It will permanently run when loaded and compute a new sample at each pixel. The result already gets averaged over time by the framework. The solution will be reset every time you change the code. To add new iterations, press the play or stop buttons in the web page. The current solution will run for 1000 samples.

The results are also *tone-mapped* and *gamma-corrected*. Typically, you do not need to change the code in the tab "Tonemapping" to solve the coursework. Modifying it might help for visual debugging in some cases, though. Tone-mapping will reduce the physical light values to a range your display can actually reproduce and that appears most plausible visually. Gamma-mapping will convert physically linear units your simulation produces into non-linear values your display expects. If you know the gamma value of your monitor, you can change the $\gamma = 2.2$ we assume to something better to see more details in light or shadows (some machines go as low as $\gamma = 1.6$ these days).

1 Let there be light! (5 points)

In the beginning you will see only the diffuse colors.



First, extend the `Material` struct to hold the information that every material in path tracing has to have to become a light source (**2 points**). For our example, the first sphere should become a source emitting $150.0 \cdot (0.9, 0.9, 0.5)$ units of light, the second should emit $150.0 \cdot (0.8, 0.3, 0.1)$ and all other object should not be light sources. Second, use this information in `getEmission` (**1 points**). You now should now see the direct light as seen below. Write two sentences about what the gamma is doing in our case (**2 points**).

2 Now bounce (10 points)

The current code is only able to render the first bounce of light between the object and the camera. We will now add multiple bounces.

To do so, first implement the function `randomDirection` to return a random direction in 3D. Parameter to this function is the `dimensionIndex` of the bounce. The i-th bounce's sampling dimension is `PATH_SAMPLE_DIMENSION`$+2\,i$. This index will later be used in advanced (e.g., Halton) sampling that proceeds differently in different dimensions.

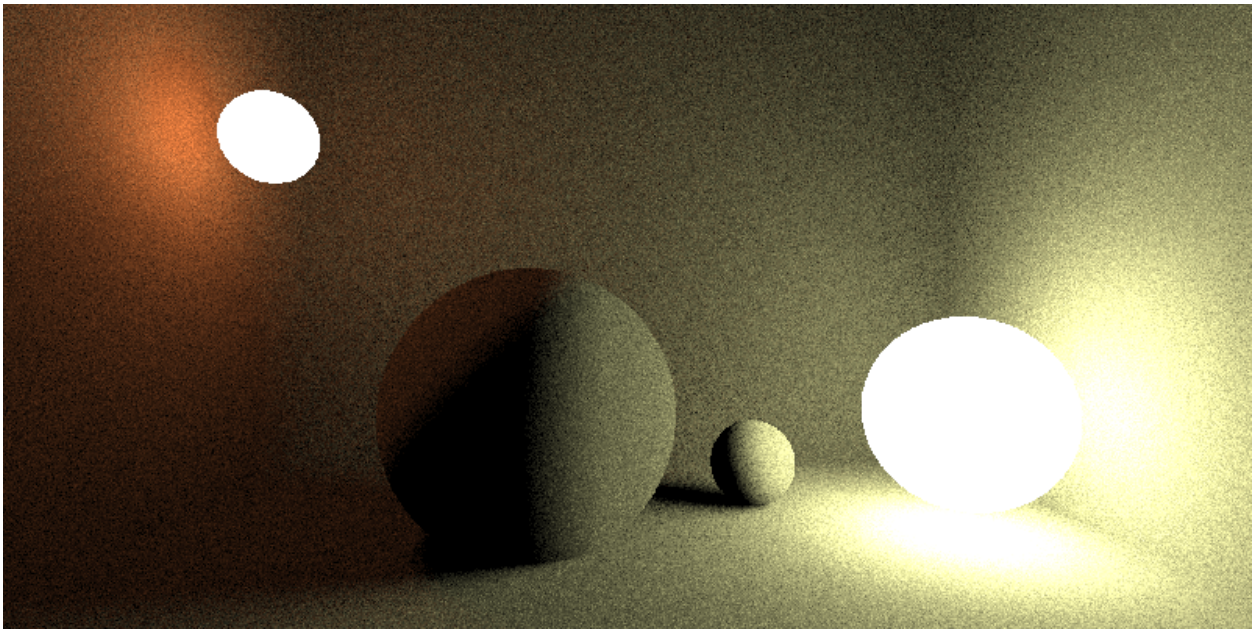
The lecture has explained how picking a random 3D point in the unit cube and normalizing it is not a valid solution. Instead, you should use the formula below to compute a vector $\omega_\text{i} = (x, y, z)$, where ξ_0 and ξ_1 are random sample coordinates in $(0, 1)$ provided by our function `sample` (**2 points**).

$$
\begin{aligned}
\theta &= \arccos(2 \cdot \xi_0 - 1) \\
\phi &= \xi_1 \cdot 2\pi \\
x &= \sin(\theta)\cos(\phi) \\
y &= \sin(\theta)\sin(\phi) \\
z &= \cos(\theta)
\end{aligned}
$$

The formula above has two logical parts, can you indentify them? Implement the formula by calling two separate functions (**1 points**) instead of one go and give them proper names (**1 points**). What would be a unit test of this, that was to involve a third function and what is that third function? (**2 points**). Implement this test and describe how it would be ran by setting a simple flag (**2 points**).

Next, you need to use this function to trace a ray in the direction ω_i. The function `intersectScene`, similar to the one used in the first coursework, is at your disposal to do so (**2 points**).

Please use the constant `const int maxPathLength` defined on the top to control the maximal length of the path you sample. At `maxPathLength = 2`, the image should look as the one below:



3 Throughput (30 points)

The current solution solves an alternative, non-physical equation that misses the reflectance and the geometric term:

$$L(\mathbf{x}, \omega_\mathrm{o}) = L_\mathrm{e}(\mathbf{x}, \omega_\mathrm{o}) + \int 0.1 \cdot L(\mathbf{y}, -\omega_\mathrm{i}) \mathrm{d}\omega_\mathrm{i}$$

What it should solve instead is an equation that include the BRDF and the geometric term:

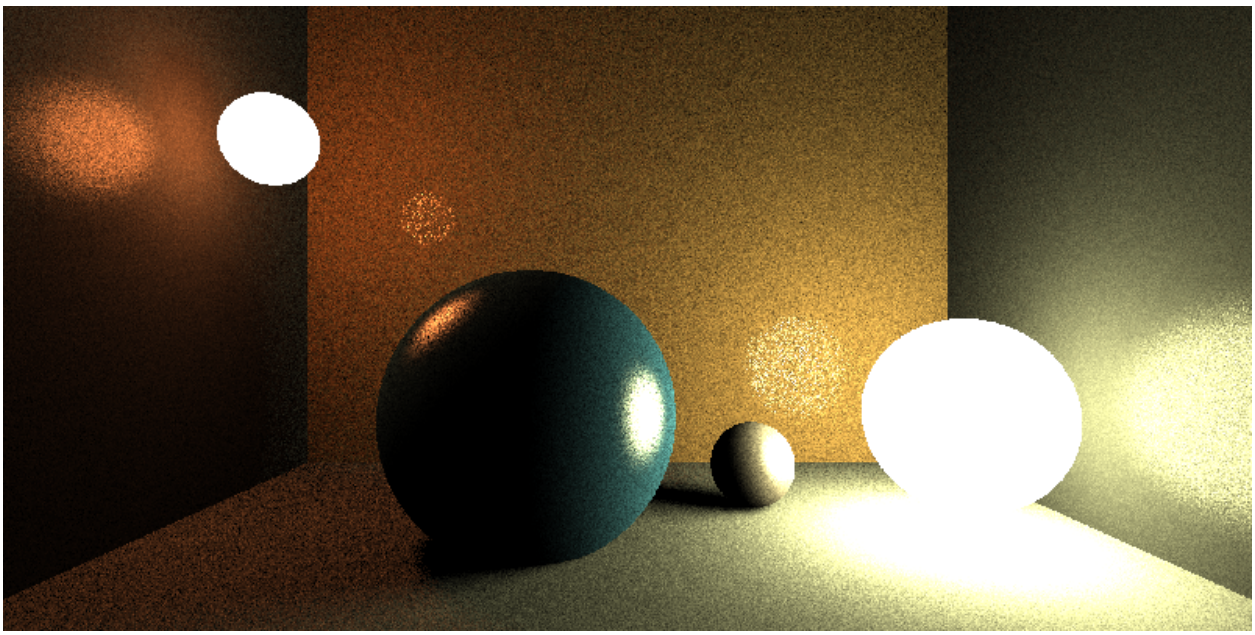
$$L(\mathbf{x}, \omega_\mathrm{o}) = L_\mathrm{e}(\mathbf{x}, \omega_\mathrm{o}) + \int f_\mathrm{r}(\mathbf{x}, \omega_\mathrm{i}, \omega_\mathrm{o}) L(\mathbf{y}, -\omega_\mathrm{i}) \cos(\theta) \mathrm{d}\omega_\mathrm{i}$$

First, implement the function `getGeometricTerm` (**5 points**) and `getReflectance` (**5 points**) using the (physically-correct) Phong BRDF. Physically-correct Phong is a variant of Phong that uses the normalization factor $^{n+2}/_{2\pi}$ to preserves energy for all values of glossiness n.

$$f_\mathrm{r}(\omega_\mathrm{i}, \omega_\mathrm{o}) = \frac{k_\mathrm{d}}{\pi} + k_\mathrm{s} \frac{n+2}{2\pi} (\langle \omega_\mathrm{o}, \mathbf{r} \rangle^+)^n$$

Implementing both methods will not yet change something, as they are not called. Second, those functions have to be multiplied up correctly to compute the throughput of every light path vertex to the camera (**10 points**). You might want to remember how the variable `weight` worked in the backward ray-tracer in coursework 1. Finally, add comments to explain your implementation of `getGeometricTerm` and `getReflectance` and how you used them to compute the throughput (**10 points**).

Solving this, a converged image will look the following:



4 Variance reduction: Sampling patterns (20 points)

Currently, we use random numbers using a simple linear congruential generator `frand` for every pixel. We have seen how quasi-random patterns, such as Halton reduce variance in the lecture. Here, we will make use of them.

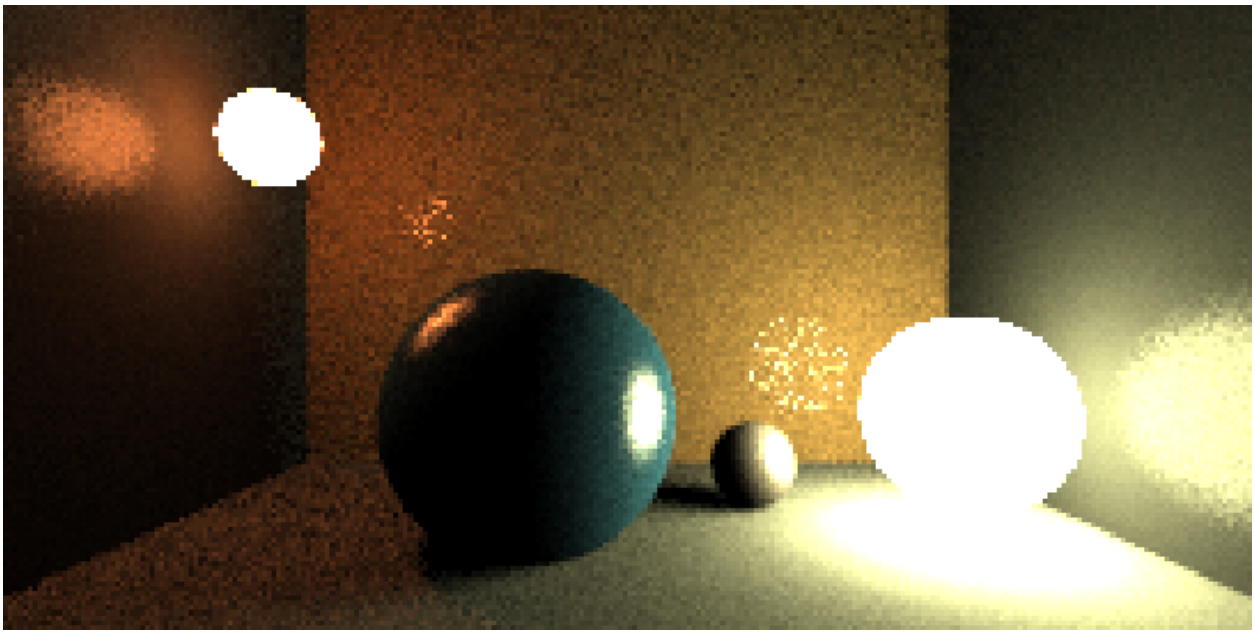
Change the implementation of `sample` to use Halton sampling (**5 points**). Note, how `sample` takes the dimension as a parameter. For the `frand` implementation, this parameter was not relevant, for Halton it will be. You might find using the existing function `prime(int index)` that returns the i-th prime number form a table useful. We do not require you to explain Halton sequences or radical inversion or Van-der-Corput sequences, as long as you use it correctly.

You will notice the effect by a quicker convergence, less noise, but structural patterns. As both methods are unbiased, the end-result will be the same. Consider comparing the images with and without the change: They should be identical.

Let $s_{i,j,k}$ be the sample in $(0, 1)$ which our function `sample` returns for dimension i, sample j and pixel k. Using the same Halton pattern $s_{i,j,k} = \mathcal{H}_{i,j}$ might produce quicker convergence, but leads to structured patterns you will find objectionable as all pixels k have the same value. Using a uniform random value $s_{i,j,k} = \xi_{i,j,k}$ avoids this, but converges slower. The lecture has mentioned *Cranely-Petterson rotation* to combine both: in dimension i all pixels use the same pattern $\mathcal{H}_{i,\cdot}$, but shifted modulo-1 by a random per-pixel and per-dimension (but not per-sample) offset $\xi_{i,k}$ as in $s(i, j, k) = \texttt{fract}(\mathcal{H}_{i,j} + \xi_{i,k})$. Add code for this improvement (**10 points**) and explain your implementation (**5 points**).

5 Anti-aliasing (10 points)

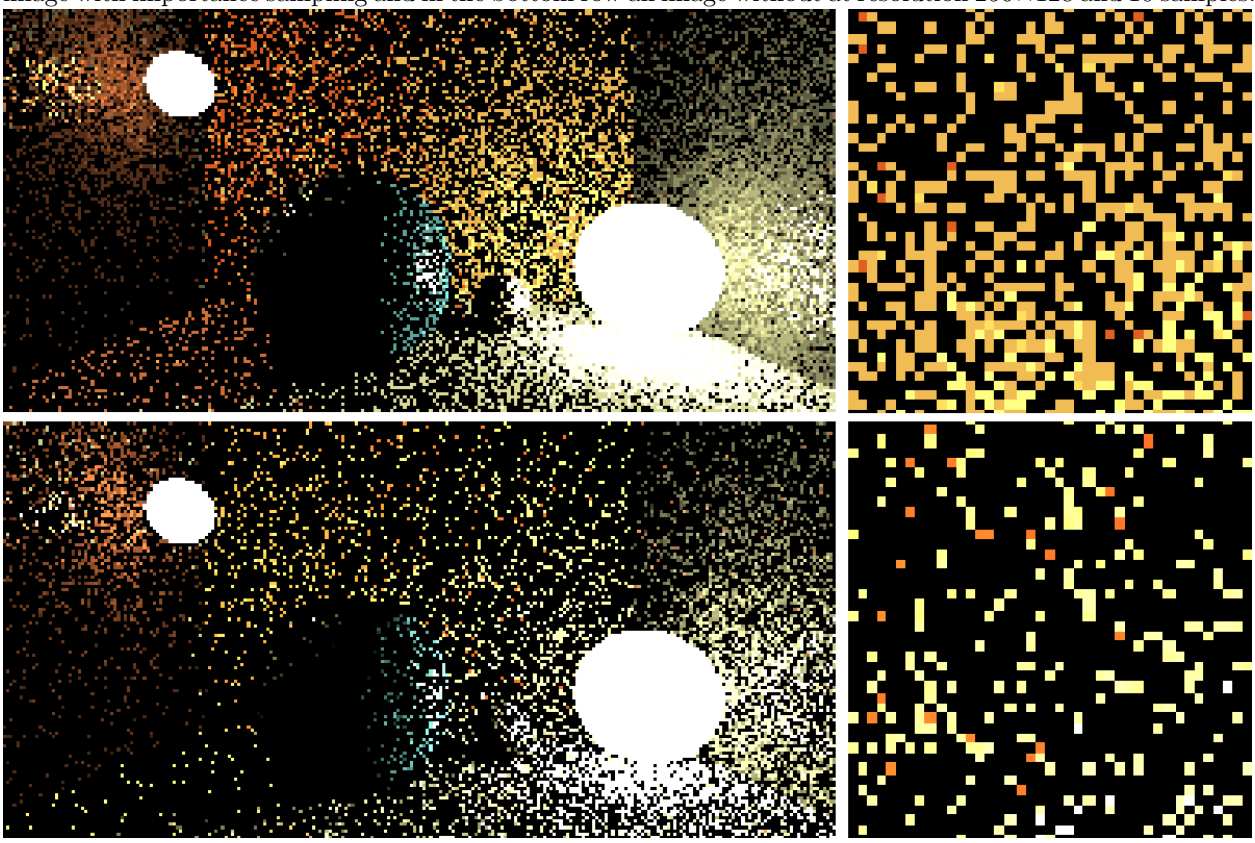
We have seen that other effects such as depth of field, motion blur and anti-aliasing can be understood as generalized sampling domains. A simple domain to tackle is the pixel domain. Add pixel sampling with a box filter to produce anti-aliasing (**5 points**) and explain your implementation (**5 points**). After adding anti-aliasing the edges should be nicely blurred as seen in the image below.



6 Importance sampling is important (20 points)

This is more open-ended and advanced questions.

Implement importance sampling for the geometric term (**10 points**). The below figure shows in the top row an image with importance sampling and in the bottom row an image without at resolution 256×128 and 10 samples.



What is the most basic job you expect such an importance sampling to do? (**1 points**). Write a comment in the importance sampling function.

What is the difference between an importance sampled result and one without it for infinitely many samples? Make sure your implementation approaches said difference for many (e.g., 10,000) samples! (**1 points**). Write a comment in the importance sampling function.

Can you demonstrate an extension to multiple light sources (emissive sphere in our case) that have very different values of emissiveness? Please implement and describe in the way you see fit (**6 points**).

Can you also propose and implement in a similar way what would be to do if some of those spheres had positions very different from the positions seen in the framebuffer (**2 points**)?

7 Motion Blur (5 points)

Add motion blur to the intergator (**5 points**).

If you give a motion of $(-3, 0, 3)$ to the sphere with index 2 and a motion of $(2, 4, 1)$ to sphere with index 3, the image will look like this: