# Coursework III:
# Path Tracing

COMP0027 Team
Tobias Ritschel, Michael Fischer, Siddhant Prakash, Chen Liu

November 30, 2023

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 and the OpenGL ES Shading Language (GLSL). Here is a quick reference for GLSL functions that you will commonly use in your coursework, e.g., `reflect` and `sin`.

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.

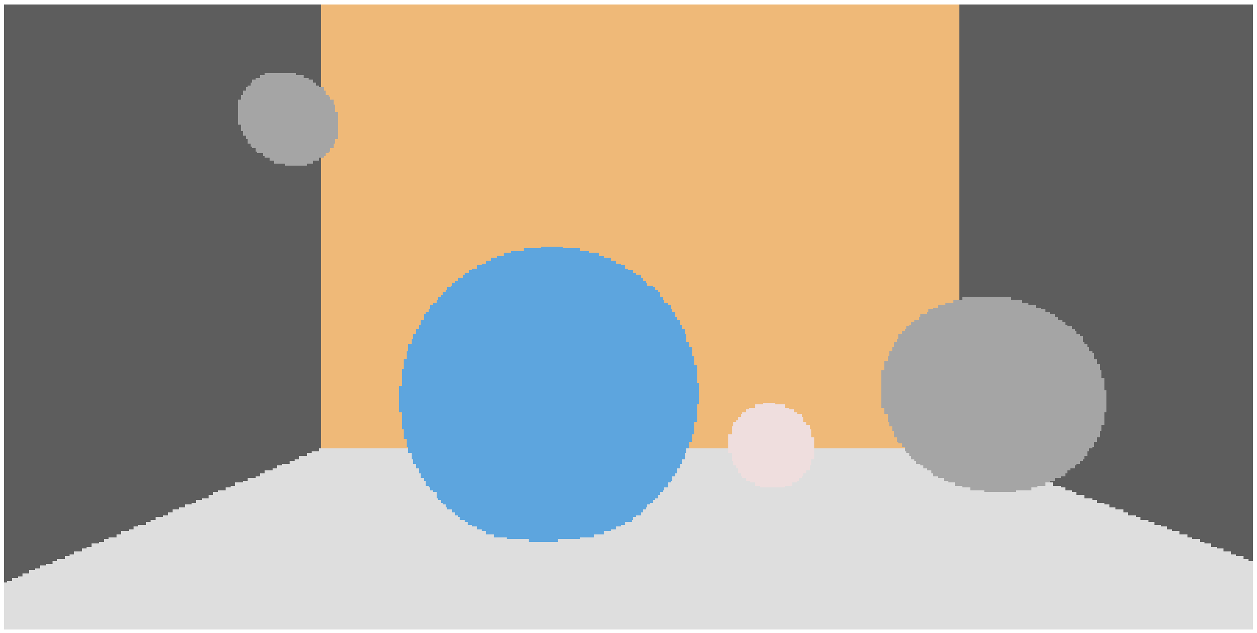
As in the previous coursework, we proceed in an artificially-low resolution for two reasons: Slow computers and in order to allow you to see individual pixels to notice subtle effects. The new revision of the framework allows to change the resolutions with a new button.

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.

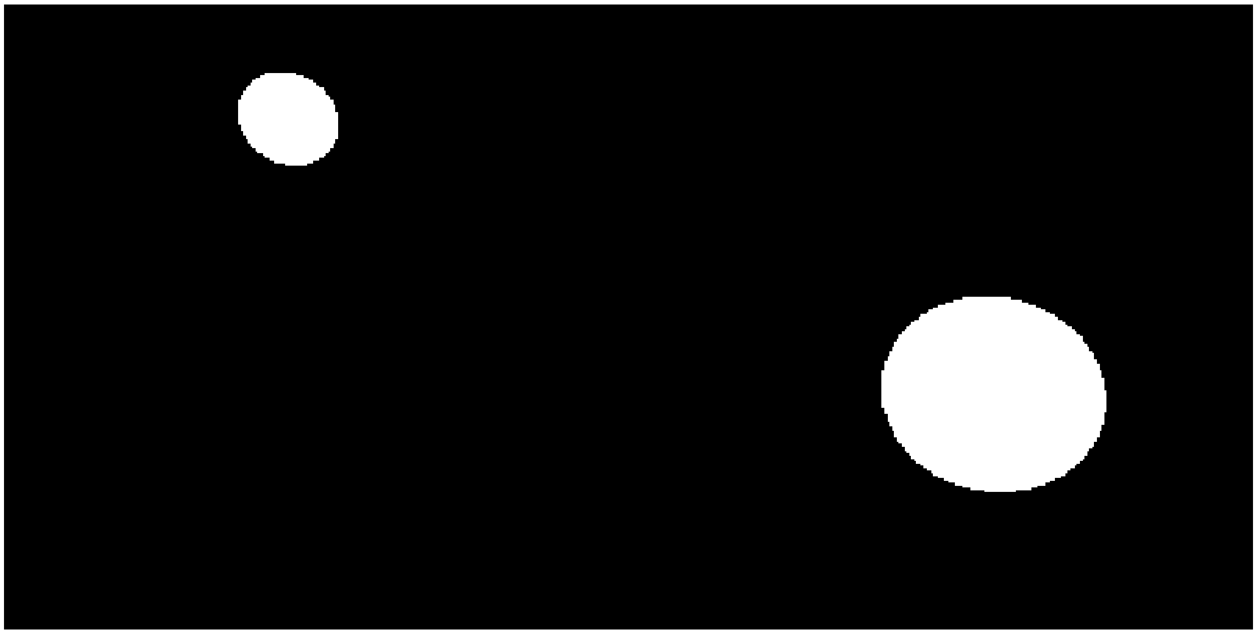
The results are also *tone-mapped* and *gamma-corrected*. Typically, you do not need to change the code in the tab "Tonemapping" to solve the coursework. Modifying it might help for visual debugging in some cases, though. Tone-mapping will reduce the physical light values to a range your display can 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 $15.0 \cdot (0.9, 0.9, 0.5)$ units of light, the second should emit $15.0 \cdot (0.8, 0.3, 0.1)$ 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 sampling that proceeds differently in different dimensions.

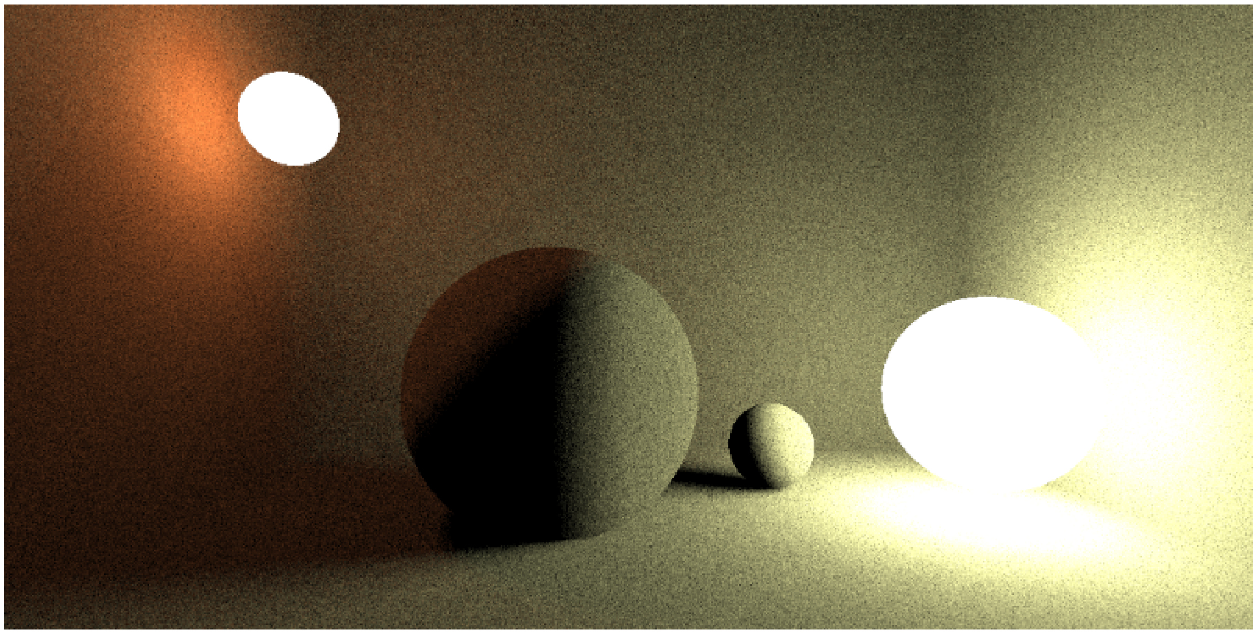
The lecture has explained how picking a random 3D point in the unit cube and normalizing it is not a valid solution. Instead, you should use the formula below to compute a vector $\omega_\mathrm{i} = (x, y, z)$, where $\xi_0$ and $\xi_1$ are random sample coordinates in $(0, 1)$ provided by our function `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 $\omega_\mathrm{i}$. 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 includes the BRDF and the geometric term:

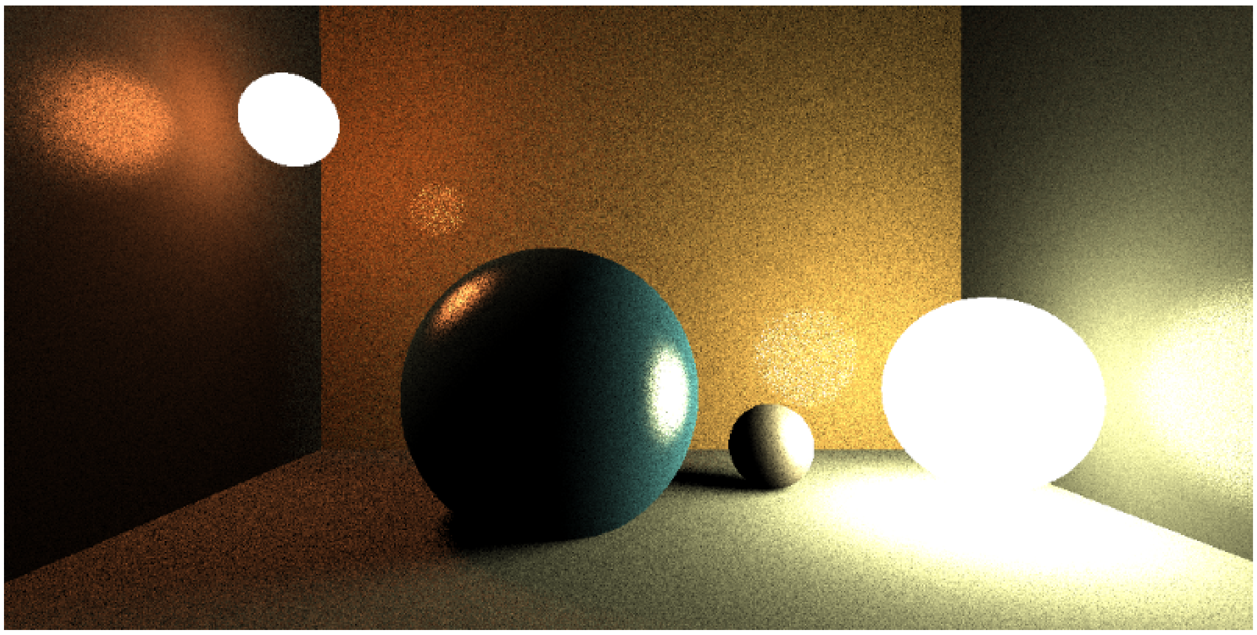
$$L(\mathbf{x}, \omega_\mathrm{o}) = L_\mathrm{e}(\mathbf{x}, \omega_\mathrm{o}) + \int f_\mathrm{r}(\mathbf{x}, \omega_\mathrm{i}, \omega_\mathrm{o}) L(\mathbf{y}, -\omega_\mathrm{i}) \cos(\theta) \mathrm{d}\omega_\mathrm{i}$$

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 preserve energy for all values of glossiness $n$.

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

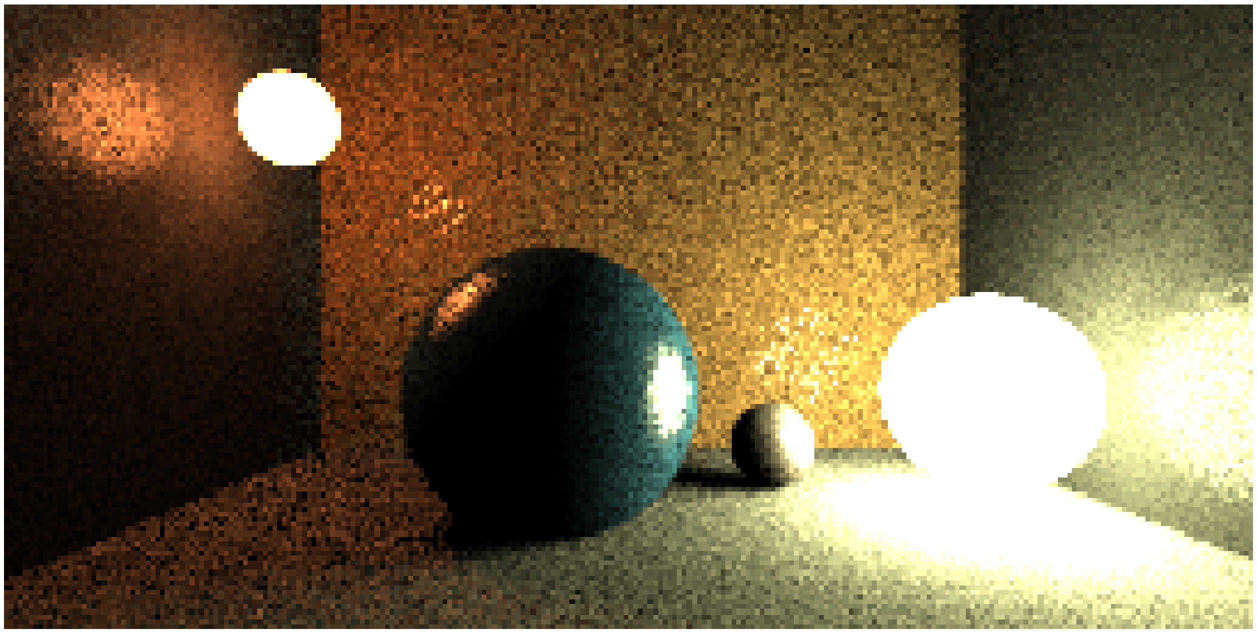
Implementing both methods will not yet change something, as they are not called. Second, those functions have to be multiplied up correctly to compute the throughput of every light path vertex to the camera (**10 points**). You might want to remember how the variable `weight` worked in the backward ray-tracer 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**).

If you have solved this correctly, the image will look like this:



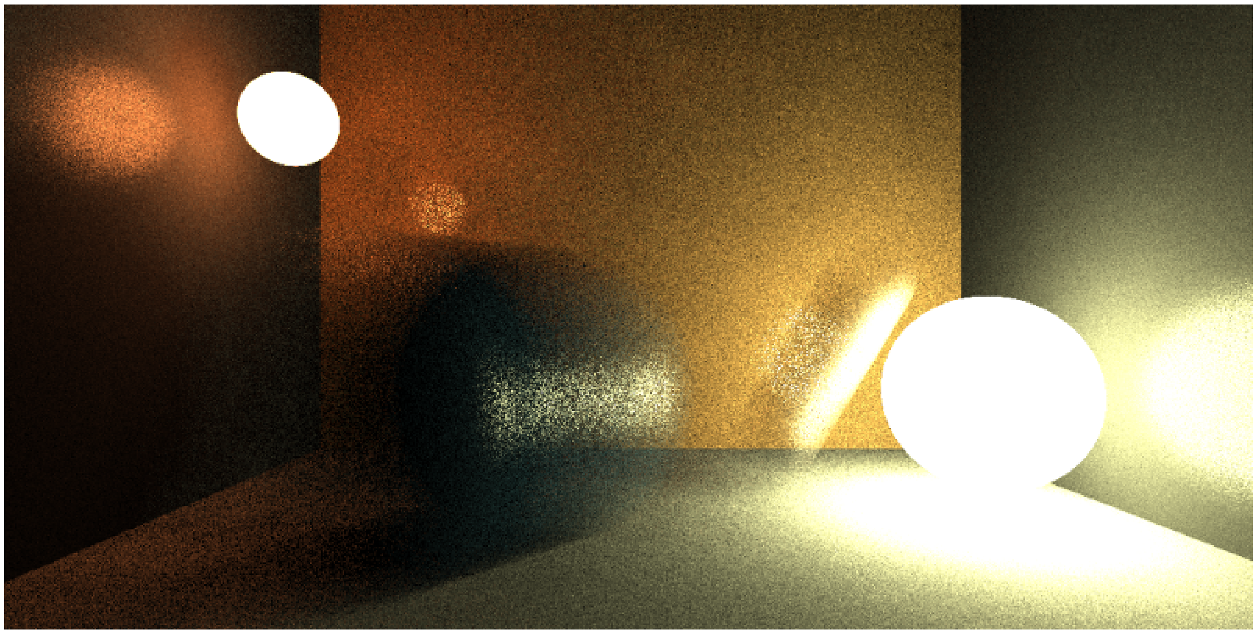
## 4 Anti-aliasing (10 points)

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.



## 5 Motion Blur (10 points)

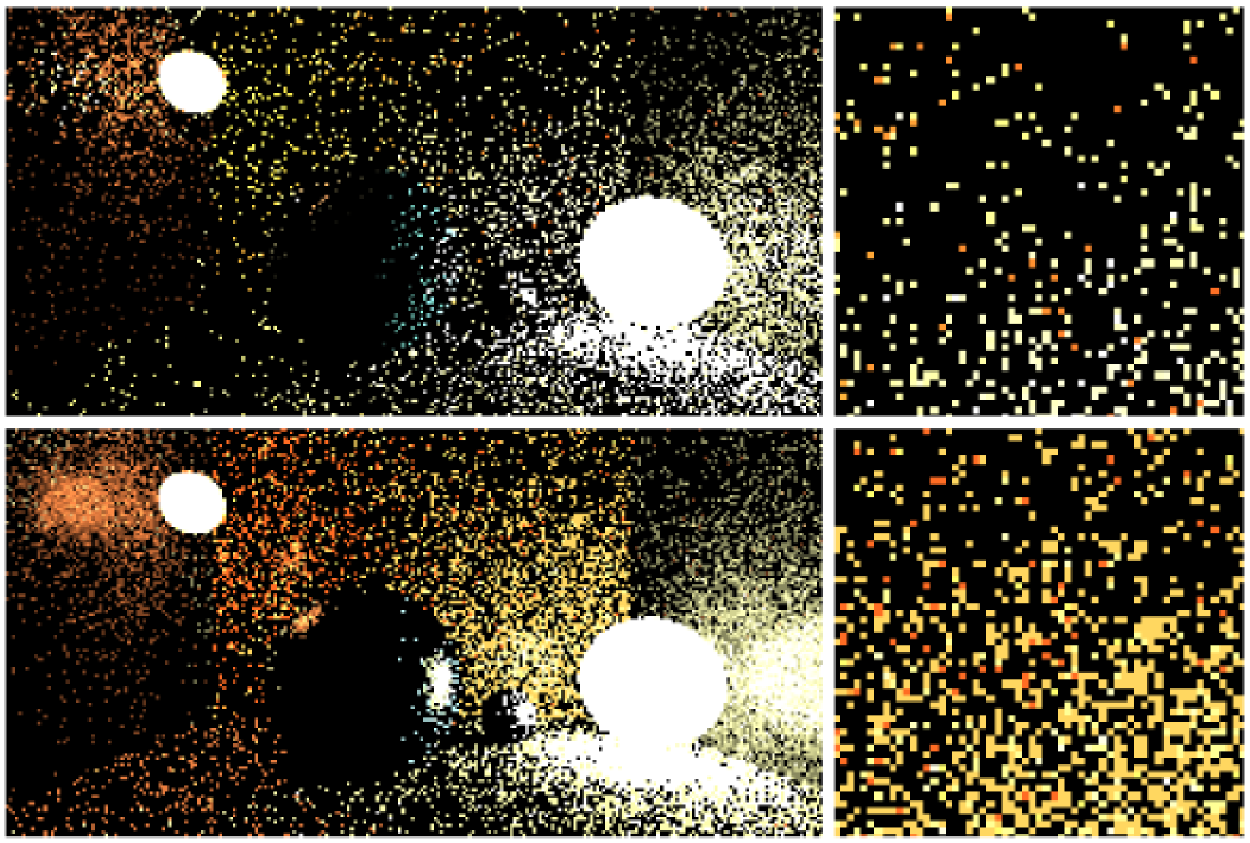
Add motion blur to the integrator (**10 points**). If you give a motion of $(-3, 0, 3)$ to the sphere with index 2 and a motion of $(2, 4, 1)$ to sphere with index 3, the image will look like this:



## 6 Variance Reduction (35 points)

This final tasks consists of more open-ended and advanced questions. We have seen in the lecture how drawing uniform-random samples $\xi$ achieves an unbiased estimator of the light transport integral, but converges slowly and exhibits significant variance. Your task here is to speed up this process via *variance reduction techniques*, while maintaining an *unbiased* estimator. You can turn the motion-blur off for this task and are allowed to write code outside the variance reduction #ifdefs, if need be.

The below figure shows what such a result could look like: for an equal sample count of 10 samples, the bottom image (with variance reduction) has already converged much further than the top image (no variance reduction, uniform random sampling). Both images are rendered at resolution 256×128.



In this task, we therefore ask you to implement **three different variance reduction techniques** of your choosing. The techniques have to be *markedly different* from one another, i.e., a different algorithm, sampling strategy or technique has to be used for each.

We will assess the correct implementation of each variance reduction technique and ask you to provide a short explanation (max. 3 sentences each) on how your implementation is working ((**24 points**), 8 each). The explanations for this task are to be handed in as a well-formatted PDF in a .zip file with your submission.

Additionally, the PDF should contain some *supporting evidence* of your algorithm's performance. We leave it up to you to choose the data format here, but the goal is to convey that a) the variance is reduced, and b) the algorithm is unbiased ((**9 points**), 3 each).

Lastly, please provide detailed comments in your code and a flag to switch between the algorithms (**2 points**).