Figure 1: A complex, textured model rendered via ray tracing with Blinn-Phong shading.

CG/task1 — Ray Tracing

Deadline: 2025-04-29 16:59:59

In order to render realistic images (such as Figure 1), it is necessary to consider the effect of light and shadows within a virtual scene. However, an exhaustive simulation of all physical components of the render equation (which models the interaction between light and matter) is typically not feasible. In order to achieve a visually pleasing result with limited resources, simplified shading models are used to determine the appearance of objects. These shading models are typically physically-inspired, but trade off being fully accurate for run-time performance. An example of such a simple and widely used shading model is *Blinn-Phong* shading.

While the act of *shading* determines how the final pixel is colored, a rendering algorithm also requires a geometric scene representation to determine which points in space are visible from what direction, and which points belong to certain objects in the scene. Although this could be solved via simple geometric primitives with easy-to-compute ray-triangle intersection, such as spheres, cuboids, or planes, these primitives are typically insufficient to model realistic virtual scenes. In order to model complex objects (such as the space ship in Figure 1), the industry standard is to use *triangle meshes* to represent geometry.

Finally, it is also necessary to move and place the camera to generate images from arbitrary positions and viewing directions.

Thus, the goal of this task is to add the following components to the existing ray tracing framework:

- Generate rays for an arbitrarily positioned and rotated camera.
- Intersect rays with spheres.
- Compute shading via the Blinn-Phong model.
- Calculate simple direct shadows via additional shadow rays.

1 Ray Tracing Basics

A ray is mathematically defined as

$$\mathbf{r}(t) = \mathbf{p} + t \cdot \mathbf{d}, \tag{1}$$

where $\mathbf{p}$ denotes the coordinates of the *ray origin* and $\mathbf{d}$ is the *ray direction*. The *ray parameter* $t > 0$ is the distance along the ray, starting from the ray origin $\mathbf{p}$ towards the ray direction $\mathbf{d}$.

The core of a ray tracing system is the generation of *primary rays* (also sometimes referred to as *camera rays* or *eye rays*), which are traced through the image plane. In order to generate a final raster image of the scene with resolution $r_x \times r_y$, we *sample* the scene through the image plane. This image plane is positioned at an offset of f in the direction $-\mathbf{w}$ in front of the camera position. The sampled region of the image plane is defined via its width w_s and height h_s. Each pixel in the output image corresponds to one sample in the image plane, which is determined by tracing a ray through it. The first object that is intersected by this ray determines the color of the output pixel. Note that the pixels in an image are typically indexed row-wise, starting top left. This means that the pixel $(0,0)$ is located on the top left of the image, while the pixel $(r_x - 1, r_y - 1)$ is located on the bottom right. The sample positions on the image plane result from dividing the image plane into a regular grid consisting of $r_x \times r_y$ cells. The rays are then cast through the *center* of each cell.

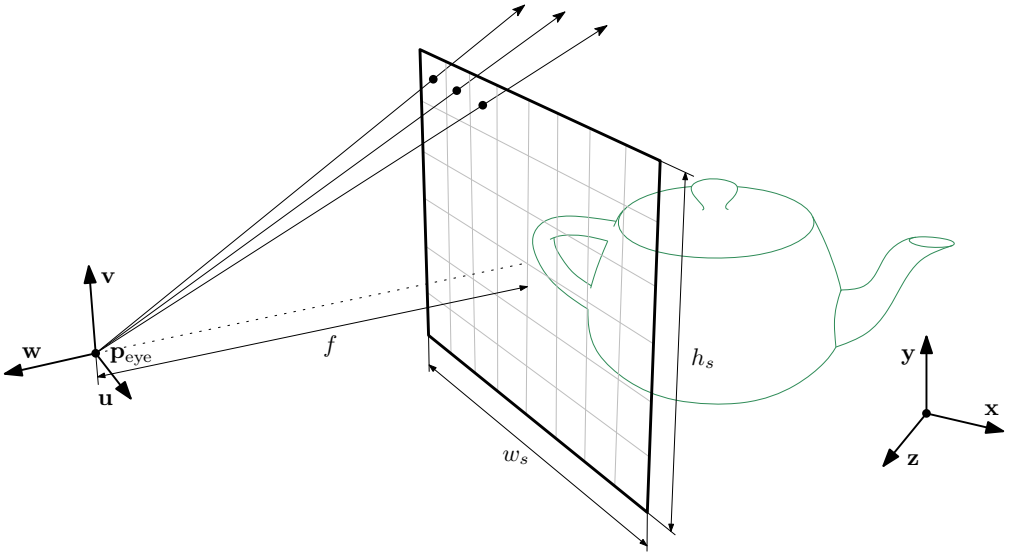


Figure 2: Primary rays are traced from the camera position $\mathbf{p}_{\text{eye}}$ towards the image plane. The rotation and position of the camera and image plane is given via the locally defined coordinate system $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ and $\mathbf{p}_{\text{eye}}$, as well as the distance to the image plane f. These axes (and their origin) can vary significantly from the scene coordinate system $\mathbf{x}, \mathbf{y}, \mathbf{z}$.

2 Camera Model

Figure 2 illustrates the generation of primary rays for an arbitrarily positioned and rotated camera, where each ray is traced from the origin of a local coordinate system $\mathbf{p}_{\text{eye}}$. The image plane is rotated along the same local coordinate system, relative to the basis vectors $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$.

The extent of the image plane is given by its width w_s and its height h_s, and the distance between the center of the image plane and the origin of the camera is given as the focal length f. As a uniform sampling of the image plane is desired, we can define

$$\frac{w_s}{h_s} = \frac{r_x}{r_y},$$

where r_x and r_y denote the resolution of the image in x and y direction.

A common way to determine the rotation and position of the camera is to first define the camera origin $\mathbf{p}_{\text{eye}}$ and a *lookat point* $\mathbf{p}_{\text{lookat}}$. These two points allow us to define the camera's forward direction later on. Additionally, we also need an *up vector* $\mathbf{v}_{\text{up}}$ to fully define the camera coordinate system, as this constrains the upwards direction of the camera.

Figure 3 shows an example for these points and vectors. With this information, we can

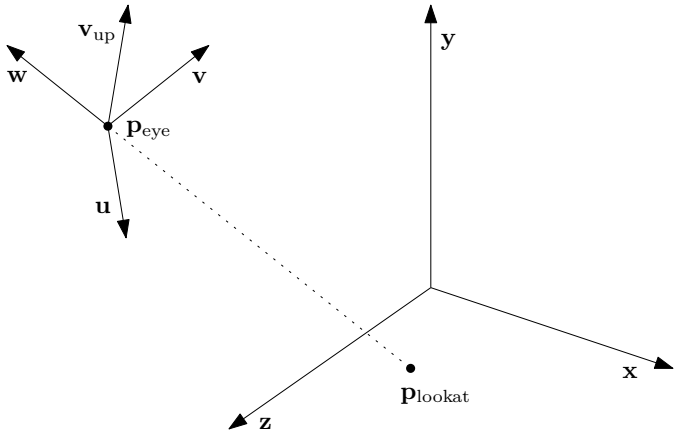


Figure 3: A reference camera coordinate system with basis vectors $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$. The camera is positioned at $\mathbf{p}_{\text{eye}}$ and rotated such that it looks towards $\mathbf{p}_{\text{lookat}}$ with a given $\mathbf{v}_{\text{up}}$ vector.

determine the basis vectors of the camera coordinate system as follows:

$$\mathbf{w} = \frac{\mathbf{p}_{\text{eye}} - \mathbf{p}_{\text{lookat}}}{\left\|\mathbf{p}_{\text{eye}} - \mathbf{p}_{\text{lookat}}\right\|}, \qquad \mathbf{u} = \frac{\mathbf{v}_{\text{up}} \times \mathbf{w}}{\left\|\mathbf{v}_{\text{up}} \times \mathbf{w}\right\|}, \qquad \mathbf{v} = \mathbf{w} \times \mathbf{u}. \tag{2}$$

3 Blinn-Phong Shading

The process of computing the color of a surface point of a visible object is commonly referred to as *shading*. Note that shading does not necessarily refer to shadows—although shadows can be part of shading, they do not have to be. Often, for simplicity, only the influence of direct illumination of the object surface is considered by using ideal, analytic point light sources. As shown in Figure 4, this simplifies the problem to calculating the portion of light that is reflected towards the viewer (in the direction $\mathbf{v}$) from the portion of light that is incoming from the direction $\mathbf{l}$ onto the surface point $\mathbf{p}$. Multiple light sources can be considered via superposition, which means summing up the contribution of each individual light source.

The reflection of light at a surface is typically modeled by a *diffuse* component and a *specular* component. The diffuse component $\mathbf{c}_d$ corresponds to the portion of the incident light that is scattered in all directions. It can be determined by Lambert's Law and only depends on the incident angle θ, which is given by the direction of the surface normal $\mathbf{n}$ in relation to the light source:

$$\mathbf{c}_d = \mathbf{k}_d \cdot \max(\cos\theta, 0), \tag{3}$$

where $\mathbf{k}_d$ is the diffuse reflection coefficient of the surface material. The specular component corresponds to the portion of the incident light that is mostly reflected into the direction reflected around the surface normal. This models effects such as shiny highlights

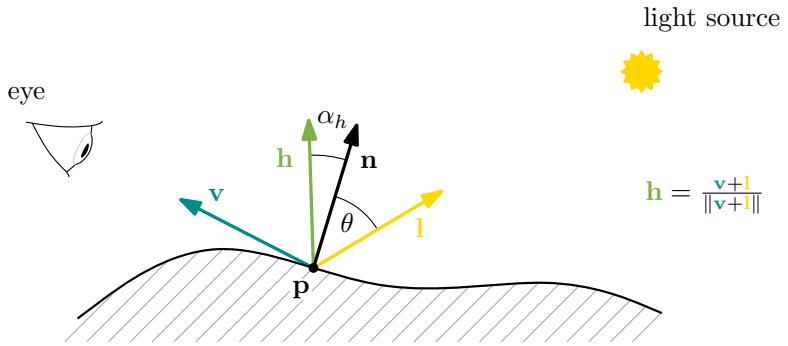


Figure 4: A local view of how a point **p** on the surface of an object is shaded by a single light source. The part of light coming from the direction **l** that is reflected towards the eye in direction **v** largely depends on the incident angle θ. The normal vector **n** denotes the orientation of the surface relative to the light source and is central to most shading models. Instead of using the reflection vector, the Blinn-Phong model introduces the halfway vector **h**, which lies symmetrically between **l** and **v**. The angle α_h between the normal **n** and the halfway vector **h** determines the intensity of the specular reflection (*e.g.*, shiny highlights or direct reflection), whereas the diffuse portion of reflected light only depends on θ.

and mirror-like effects. In the Blinn-Phong model, the specular component $\mathbf{c}_s$ that is reflected in the direction **v** is estimated based on the angle α_h between **h** and **n**:

$$\mathbf{c}_s = \begin{cases} \mathbf{k}_s \cdot (\max(\cos\alpha_h, 0))^m & \text{if } \cos\theta > 0 \\ 0 & \text{else} \end{cases}. \tag{4}$$

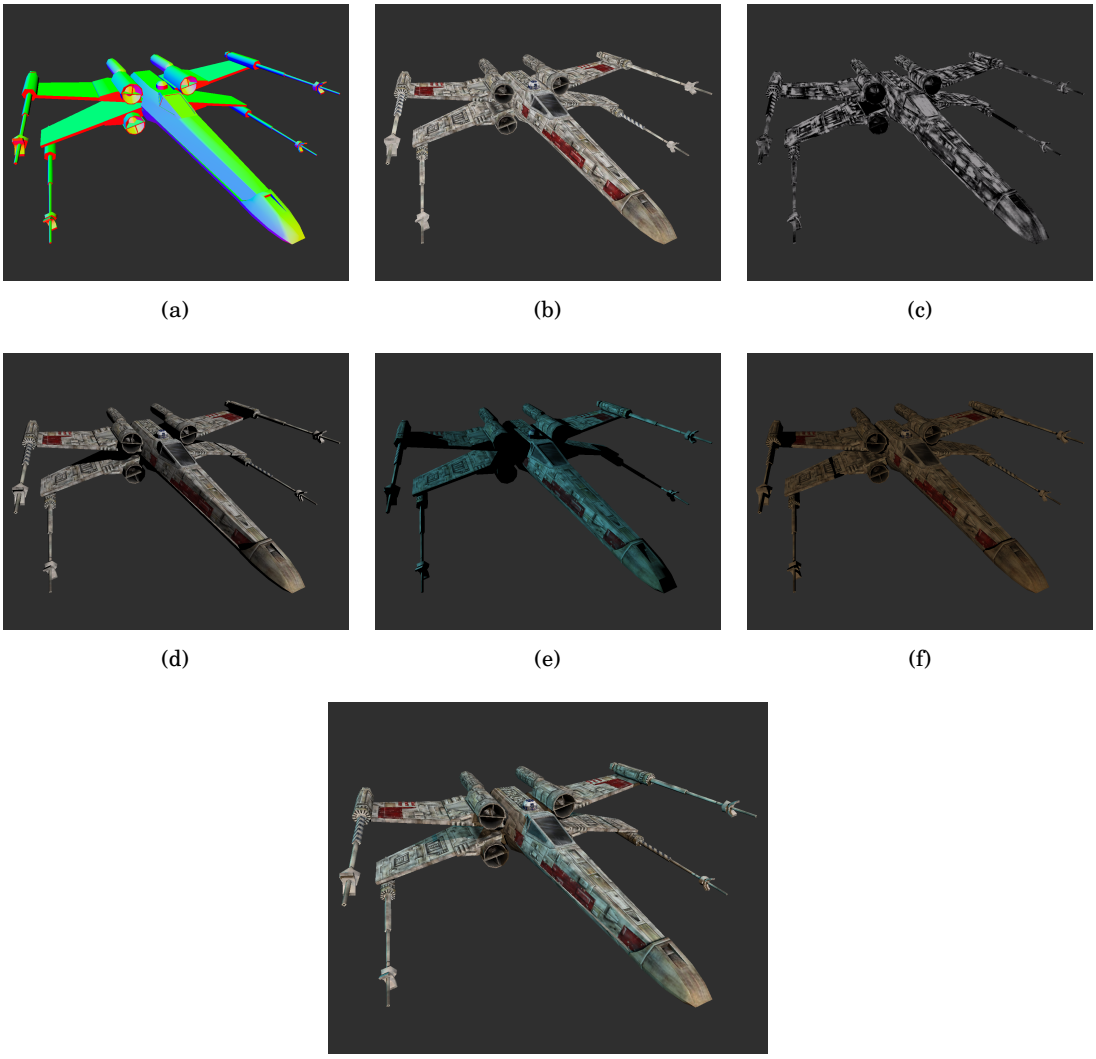
Here, $\mathbf{k}_s$ is the specular reflection coefficient of the material. The factor m models the sharpness of the specular reflection and corresponds to the shininess of the object. For larger m, the highlights are stronger but smaller, whereas smaller values of m result in larger but weaker highlights.

Hint: Given two vectors **a** and **b** and the angle θ in between them, the following relationship holds:

$$\cos(\theta) = \frac{\langle \mathbf{a}, \mathbf{b} \rangle}{\|\mathbf{a}\| \|\mathbf{b}\|}, \tag{5}$$

where $\langle \cdot, \cdot \rangle$ denotes the dot-product. Conveniently, if **a** and **b** are unit vectors, the expression above simplifies to $\cos(\theta) = \langle \mathbf{a}, \mathbf{b} \rangle$.

Figure 5 illustrates the individual components of the shading model for the rendered image shown in Figure 1.



(a) (b) (c)

(d) (e) (f)

(g)

Figure 5: Components of the shading computation. (a) Surface normal vectors, illustrated by plotting their components as colors. (b), (c) Diffuse and specular reflection coefficient, which consist of three values each for separate color channels. These coefficients vary with the texture of the model across the surface. (d), (e), (f) Individual contribution of each of the three light sources in the scene. (g) Final rendered image.

4 Intersection with a Sphere

A sphere with center $\mathbf{c}$ and radius $r > 0$ can be described as the set of all points $\mathbf{x}$ that satisfy

$$\|\mathbf{x} - \mathbf{c}\| = r, \tag{6}$$

which means the set of all points at a distance r from the center $\mathbf{c}$.

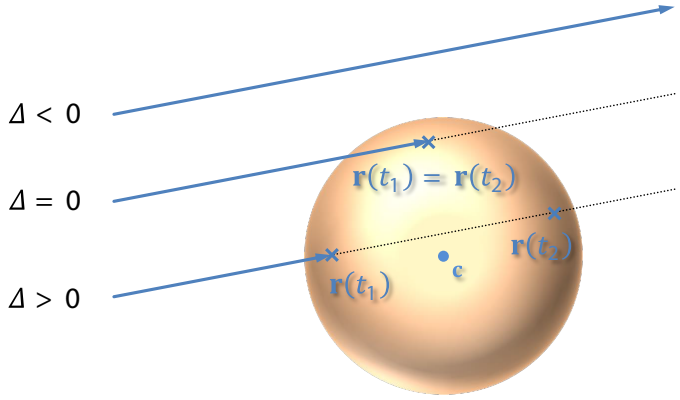


Figure 6: Computation of the intersection of a ray $\mathbf{r}(t)$ with a sphere: The discriminant Δ determines whether the ray intersects the sphere at one, two, or no points.

The intersection points of a ray with a sphere are exactly those points that lie on the ray $\mathbf{r}(t) = \mathbf{p} + t \cdot \mathbf{d}$ and also satisfy the sphere equation:

$$\begin{aligned} \|\mathbf{r}(t) - \mathbf{c}\| &= r, \\ \|\mathbf{p} + t \cdot \mathbf{d} - \mathbf{c}\| &= r, \\ \|\mathbf{p} + t \cdot \mathbf{d} - \mathbf{c}\|^2 &= r^2, \\ \langle \mathbf{p} + t \cdot \mathbf{d} - \mathbf{c},\ \mathbf{p} + t \cdot \mathbf{d} - \mathbf{c} \rangle &= r^2. \end{aligned}$$

This is a quadratic equation in t:

$$At^2 + Bt + C = 0, \tag{7}$$

with

$$\begin{aligned} A &= \langle \mathbf{d}, \mathbf{d} \rangle, \\ B &= 2 \langle \mathbf{p} - \mathbf{c}, \mathbf{d} \rangle, \\ C &= \langle \mathbf{p} - \mathbf{c}, \mathbf{p} - \mathbf{c} \rangle - r^2. \end{aligned}$$

By solving this equation for t using the quadratic formula, we obtain the ray parameter of the intersection points:

$$t_{1,2} = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}. \tag{8}$$

The discriminant

$$\Delta = B^2 - 4AC$$

determines, as illustrated in Figure 6, whether there exists one, two, or no solutions for t. If $\Delta > 0$, the ray intersects the sphere at two points. If $\Delta = 0$, the ray touches the sphere at exactly one point (the two intersection points coincide). If $\Delta < 0$, the ray does not intersect the sphere at any point.

5 Super-Sampling Anti-Aliasing (SSAA)

When rendering a scene onto a discrete pixel grid, *aliasing* artifacts can occur if the sampling rate is too low to capture high-frequency details. These artifacts often manifest as jagged edges, moiré patterns, or flickering when the camera or objects move.

A straightforward approach to reducing aliasing is *Super-Sampling Anti-Aliasing* (SSAA). Instead of computing a single color per pixel, SSAA renders the scene at a higher sampling rate by generating multiple rays per pixel. These rays are offset within each pixel, as shown in Figure 7 for example. The colors from these n^2 subpixel samples are then combined.

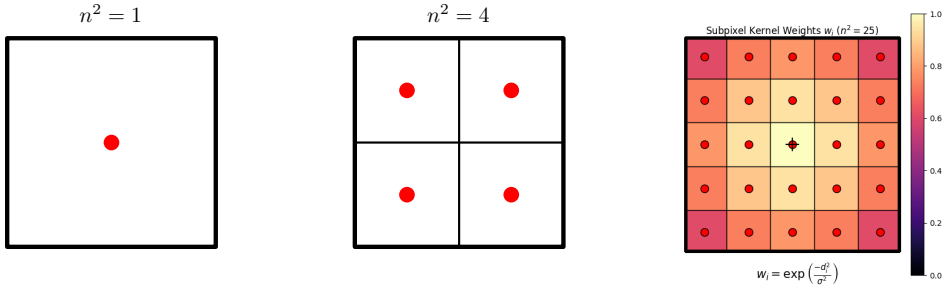


Figure 7: The effect of SSAA, considering a single pixel: Left: If no SSAA is applied, we trace our rays through the center point of each pixel (red dots). Middle: Otherwise, we evenly divide the pixel horizontally and vertically into subpixels and evaluate their respective center points. Your implementation should support any number of divisions in pixel height and width (height and width are always divided by the same integer), meaning any number of subpixels $N = n^2$, where $n \in \mathbb{N}$, $n \geq 1$. Right: We weight the resulting colors by a Gaussian Kernel with its mean at the pixel center.

Straightforward solutions perform this combination of sampled values by simply averaging their individual components. Formally, the final pixel color is then computed as

$$\mathbf{C}_{\text{final}} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{C}_i, \tag{9}$$

where $N = n^2$ is the total number of samples per pixel and $\mathbf{C}_i$ is the computed color for the i-th sample.

More advanced implementations introduce a kernel function for determining how much each sample contributes to the final color, based on the position inside the pixel. This method improves the final quality by giving more weight to subpixel colors that are closer to the center, where the sample is typically more representative. As a result, contributions from samples near the edges are down-weighted, leading to smoother edges and more faithful color reproduction. For doing so, the Euclidean distance from the pixel center is computed for each sample:

$$d_i = \sqrt{(sub\_x - x)^2 + (sub\_y - y)^2},$$

where (sub_x, sub_y) are the subpixel coordinates and (x, y) is the pixel center. The distance from the center of any pixel to its corner would be equal to $\frac{\sqrt{2}}{2}$ and represents an upper bound for d_i. Based on this distance, the weight for this sample is calculated using a kernel. For this exercise, this is done by using the following Gaussian weighting function:

$$w_i = \exp\left(\frac{-d_i^2}{\sigma^2}\right),$$

where exp(·) refers to computing the exponential function e^x (e is Euler's number), which you can simply use as `std::exp()` in your code. Here, σ^2 reflects the variance of the kernel (thus, σ is the standard deviation).

The final pixel color is then computed as the weighted sum of the subpixel colors $\mathbf{C}_i$, normalized by the sum of the weights:

$$\mathbf{C}_{\text{final}} = \frac{\sum_{i=1}^{N} w_i \mathbf{C}_i}{\sum_{i=1}^{N} w_i}, \tag{10}$$

where N is the total number of subpixel samples.

Since each pixel color is now computed by sampling multiple points within the pixel area, SSAA can significantly reduce high-frequency aliasing artifacts in the final image, resulting in smoother edges and increased overall quality. But, due to the increased number of rays, this leads to a significant performance overhead. Thus, in time-sensitive applications, more efficient techniques are preferred, such as selective supersampling near edges (MSAA). However, SSAA still yields better results compared to these performance focused methods in most cases, as it renders the entire scene at higher resolution.

6 Assignment Tasks

The goal of this exercise is to develop a simple ray tracing system with an arbitrarily oriented camera and Blinn-Phong shading. The provided framework already loads a scene via a JSON config file and generates the resulting image as a PNG file. Furthermore, the framework already contains functions that determine intersections between rays and planes, triangles and cones. This will allow you to test components individually, even if your sphere intersection tests are not functional yet. The path to the scene config JSON is passed as a command line argument to the program. You can find various test scenes in the subdirectory of your repository «`data/task1/`». If the scene renders successfully without errors, the framework produces a file called «`<config-name>.png`» in «`output/task1/`». You can use the parameters `-t` or `--num-threads` <*count*> to use multiple threads simultaneously to speed up the rendering of the output image. With `--num-threads` you can specify the number of threads, while `-t` uses as many threads as there are logical cores on your system.

6.1 Build and Submission

You need to create your repository on `https://courseware.icg.tugraz.at/`, after which you can access it on `https://assignments.icg.tugraz.at/`. You will need to use git to pull and push code in your repository. Your repository comes configured with a master branch, which you can use to freely develop and experiment with changes.

To obtains points for the exercise, you *must* create a `submission` branch and push your code into this branch. This will also trigger an automated test, for which you can view build logs, output logs and rendered images in the CI/CD section of the gitlab webinterface. **We will only grade solutions that have been pushed to the submission branch of your repository!** To build the framework, please follow the build instructions in the «`README.md`» file within your repository.

6.2 Ray Generation for arbitrary Cameras (4 Points)

Your first task is to implement the function

```
void render(image2D<float3>& framebuffer,
            int left, int top, int right, int bottom,
            const Scene& scene,
            const Camera& camera,
            const Pointlight* lights,
            std::size_t num_lights,
            const float3& background_color,
            int max_bounces,
            int ssaa_samples_per_pixel,
            float std_dev)
```

in «task1.cpp». This function is responsible for generating rays of a subsection of the framebuffer that is defined by `left`, `top`, `right` and `bottom` such that an image of the `scene` is generated. The reason that the ray generation is split up into regions of the framebuffer is that it allows for parallel computation, so make sure to consider the region of the framebuffer correctly. The main procedure of this function is described in Section 1 and Section 2, and you will need to trace one ray through each pixel of the framebuffer. Each ray needs to be traced exactly through the center of each cell in the image plane. The struct `camera` contains the necessary camera parameters `camera.w_s` (width of the image plane), `camera.f` (focal length) as well as the position of the camera `camera.eye`, the lookat point `camera.lookat` and the up vector (`camera.up`, see Section 2). Within the `render()` function, you need to call `scene.findClosestHit()` in order to determine the (nearest) intersection point of the ray. `scene.findClosestHit()` already returns correct results for cones, triangles and planes; for spheres to work, you have to implement the Ray-Sphere intersection (described below).

If a relevant intersection point exists, it needs to be passed to the `shade()` function. The `lights` parameter is a pointer to the first element of an array of `num_lights` light sources, which is further used for shading computation and will need to be passed to `shade()`. The `background_color` argument contains the color that should be written into the framebuffer if no object is visible for a specific pixel. The arguments `ssaa_samples_per_pixel` and `std_dev` will be discussed in a later section, covering the implementation of SSAA and are only used for this feature; these can be ignored when not implementing SSAA. The argument `max_bounces` is only relevant for the bonus task.

Hint: You can use image differences to detect small offsets to the reference implementation.

6.3 Ray-Sphere Intersection (2 Points)

Implement the function in «task1.cpp»

```
bool intersectRaySphere(const float3& p,
                        const float3& d,
                        const float3& c,
                        float r,
                        float& t)
```

which computes the intersection of a ray with a sphere (*cf.* Section 4). The function receives the origin and direction of the ray in the parameters `p` and `d`, and the center and radius of the sphere in `c` and `r`. If the ray intersects the sphere, the function should write the ray parameter of the intersection point to `t` and return `true`, otherwise `false`. If there are two intersection points, the one closer to the ray origin should be returned.

6.4 Blinn-Phong Shading (3 Points)

For shading, you need to implement

```
float3 shade(const float3& p,
             const float3& d,
             const HitPoint& hit,
             const Scene& scene,
             const Pointlight* lights,
             std::size_t num_lights)
```

in «task1.cpp». As described in Section 3, this function should compute (and return) the color for a given surface point, which is given as the argument `hit`. The arguments `p` and `d` again correspond to the ray origin $\mathbf{p}$ and ray direction $\mathbf{d}$ of the ray that intersected the point `hit`. The position of the intersected point can be accessed via `hit.position`, and the normal vector of this surface point can be accessed via `hit.normal`. The light sources are given in the array `lights` with `num_lights` elements, and contain `position` and `color` members within the `Pointlight` structs. Given the diffuse and specular coefficients in `hit.k_d` and `hit.k_s`, as well as the shininess `hit.m`, you can calculate the shading of the surface at the given point for all the provided light sources and return the final color.

6.5 Shadows (3 Points)

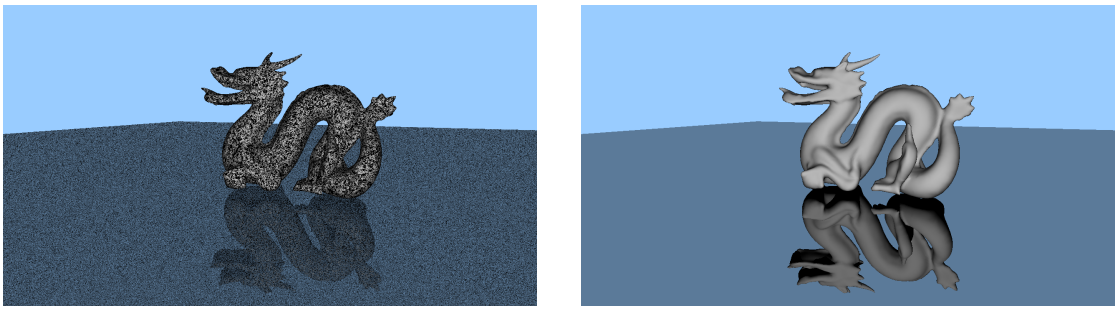
A simple way to determine if any point is directly in light of a particular light source is to cast an additional shadow ray. This shadow ray is cast from the point in question (which will be the surface location of a point that we are currently shading) towards the direction of the light source, and the objective is to determine if there is any intersection point between the origin of the shadow ray and the position of the light source.

To implement these simple direct shadows, you need to extend the function `shade()` such that the shading computation considers if a point is in shadow of a particular light source or not. To achieve this, you can cast an additional shadow ray using `scene.intersectsRay()` to determine if a given ray in direction of the light source is blocked by another object or not. In addition to the ray origin and direction of this shadow ray, you also need to provide the arguments `t_min` and `t_max`, which denote the lower and upper bound of the interval for which the intersection test should be performed. In other words, the parameter t in the following equation is bounded:

$$\mathbf{r}(t) = \mathbf{p} + t \cdot \mathbf{d} \qquad \text{subject to} \qquad t \in [t_{min},\ t_{max}]. \tag{11}$$

Due to limited precision of floating point operations, the determined intersection point between any ray and triangle typically does not lie exactly in the plane of the triangle. This can lead to artifacts such as shown in Figure 8. In this case, the intersection point itself was detected as an object blocking the light.

Therefore, you can not just directly use the intersection point `hit.position` as the origin of your shadow ray, but you first need to move it by a distance `epsilon` (defined in `task1.cpp`) in the direction of the surface normal vector. Note that our choice of `epsilon` leads to some black pixels in the reference implementation. This way, you can ensure that the origin of the shadow ray is not behind the shaded surface point due to floating point inaccuracies, and thus does not cast a shadow onto itself.



(a) (b)

Figure 8: (a) Due to floating point inaccuracies during intersection testing, the points for which shading is evaluated do not exactly lie on the object surface, but slightly in front or behind the surface. If this happens, the shading points can potentially lie in shadow of their own object, which can result in a lot of output noise. (b) By moving the intersection point by a distance ϵ in the direction of the surface normal you can ensure that the intersection point either lies on or slightly outside the object surface.

6.6 Super-Sampling Anti-Aliasing (SSAA) (4 Points)

For implementing anti-aliasing, you need to extend the function

```
void render(image2D<float3>& framebuffer,
            int left, int top, int right, int bottom,
            const Scene& scene,
            const Camera& camera,
            const Pointlight* lights,
            std::size_t num_lights,
            const float3& background_color,
            int max_bounces,
            int ssaa_samples_per_pixel,
            float std_dev)
```

in «`task1.cpp`». As described in Section 5, aliasing artifacts in computer graphics occur when rendering at a resolution that is too low to accurately capture high-frequency details in a scene. Therefore, you need to modify your existing rendering loop, such that it generates multiple rays per pixel instead of a single one. Here, you have to consider the `ssaa_samples_per_pixel` argument, which specifies the number of samples per pixel. For an `ssaa_samples_per_pixel` of N, each pixel is subdivided into $n \times n$ subpixels, resulting in a total of $N = n^2$ samples per pixel. These sample rays should be evenly distributed within the output pixel area.

Hint: You can expect $n = \sqrt{\texttt{ssaa\_samples\_per\_pixel}}$ to be an integer.

After computing the color for each sample, the final pixel color is determined as the average of all sampled colors, as shown in Equation 9.

To improve the quality of the output image, we additionally implement the Gaussian Kernel, as denoted in Equation 10. In this case, use the provided value for the standard deviation from the function parameters: `std_dev`.

For processing the sampled pixel values, only one implementation is required; however, if you choose to implement both versions, ensure that only the kernel-based approach is executed in your submission (*e.g.*, by commenting out the other version).

6.7 Bonus: Recursive Ray Tracing (3 Points)

Your goal for the (optional) bonus task is to implement recursive ray tracing, which sends out rays repeatedly if completely reflective surfaces are intersected. To do this, you need to modify the `render()` function such that surfaces with infinitely high shininess (`hit->m`$= \infty$)[1] are treated as mirror surfaces. For such surfaces, you should compute the

[1] You can use `std::isinf()` to test this.

reflected direction by reflecting the incident view direction **v** across the surface normal **n**, and then trace an additional ray in this direction. As with the shadow rays, you again need to offset your new ray origin by a small `epsilon` to prevent self-intersections due to floating point inaccuracies.

The final color of the surface point is then computed as the color of the recursively traced ray weighted by the specular reflection coefficient `hit.k_s`. To prevent an endless recursion (for example, if two mirrors face each other), you should only consider a maximum of `max_bounces` recursions. If the last ray also hits a completely reflective surface, simply call the `shade()` function for this surface (because `hit.m`$=\infty$, the highlight will be infinitely small).