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

CG/task1 — Ray tracing

In order to render realistic images (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 *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 a triangle mesh and planes.
- Compute shading via the 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.

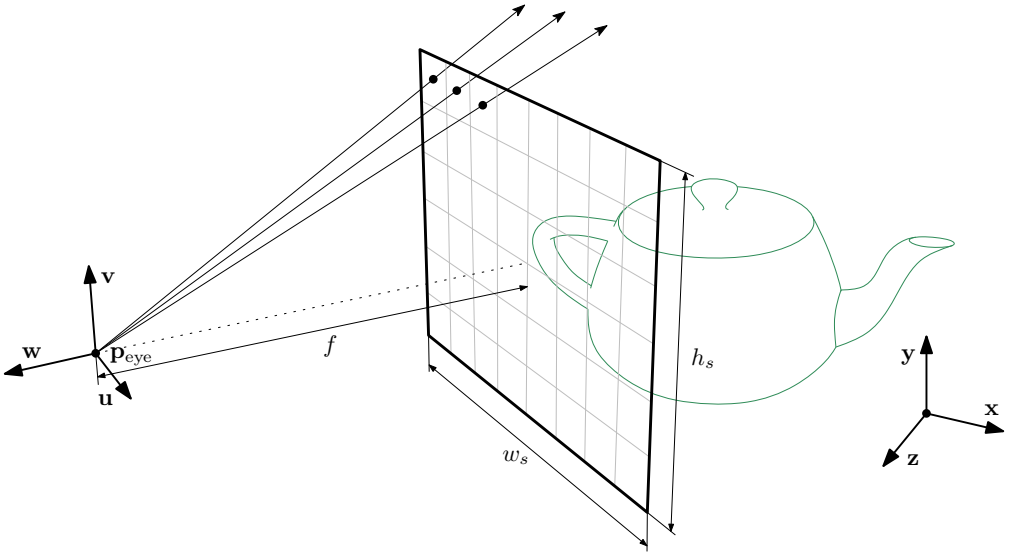


Figure 2: Primary rays are traced from the camera position towards the image plane. The rotation and position of the camera and image plane is given via the locally defined coordinate system $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ and $\mathbf{p}_{\text{eye}}$, as well as the distance to the image plane 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

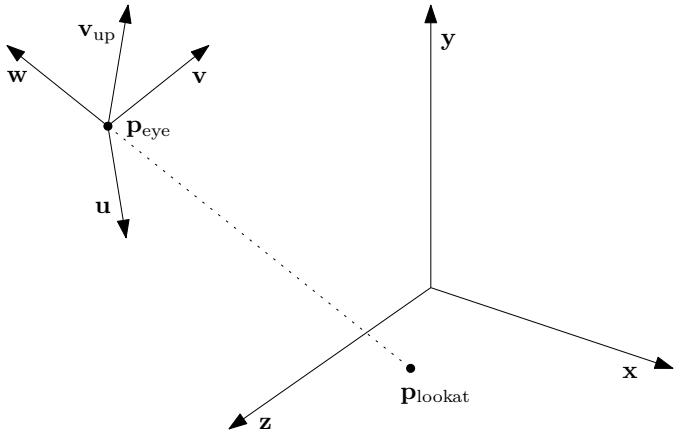


Figure 3: A reference camera coordinate system with basis vectors $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$. The camera is positioned at $\mathbf{p}_{\text{eye}}$ and rotated such that it looks towards $\mathbf{p}_{\text{lookat}}$ with a given $\mathbf{v}_{\text{up}}$ vector.

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 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 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:

$$c_d = k_d \cdot \max(\cos\theta, 0); \tag{3}$$

k_d is the diffuse reflection coefficient of the surface material. The specular component

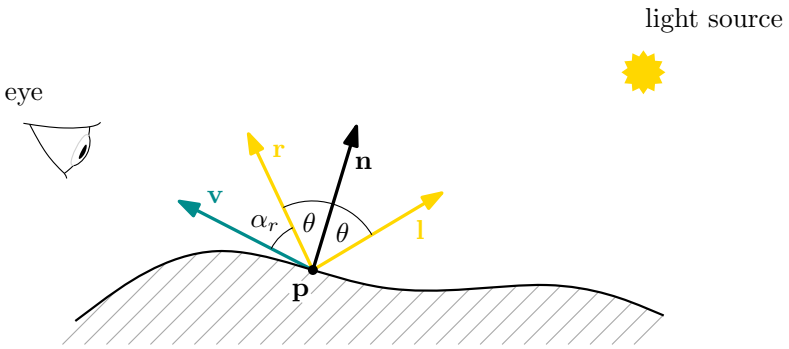


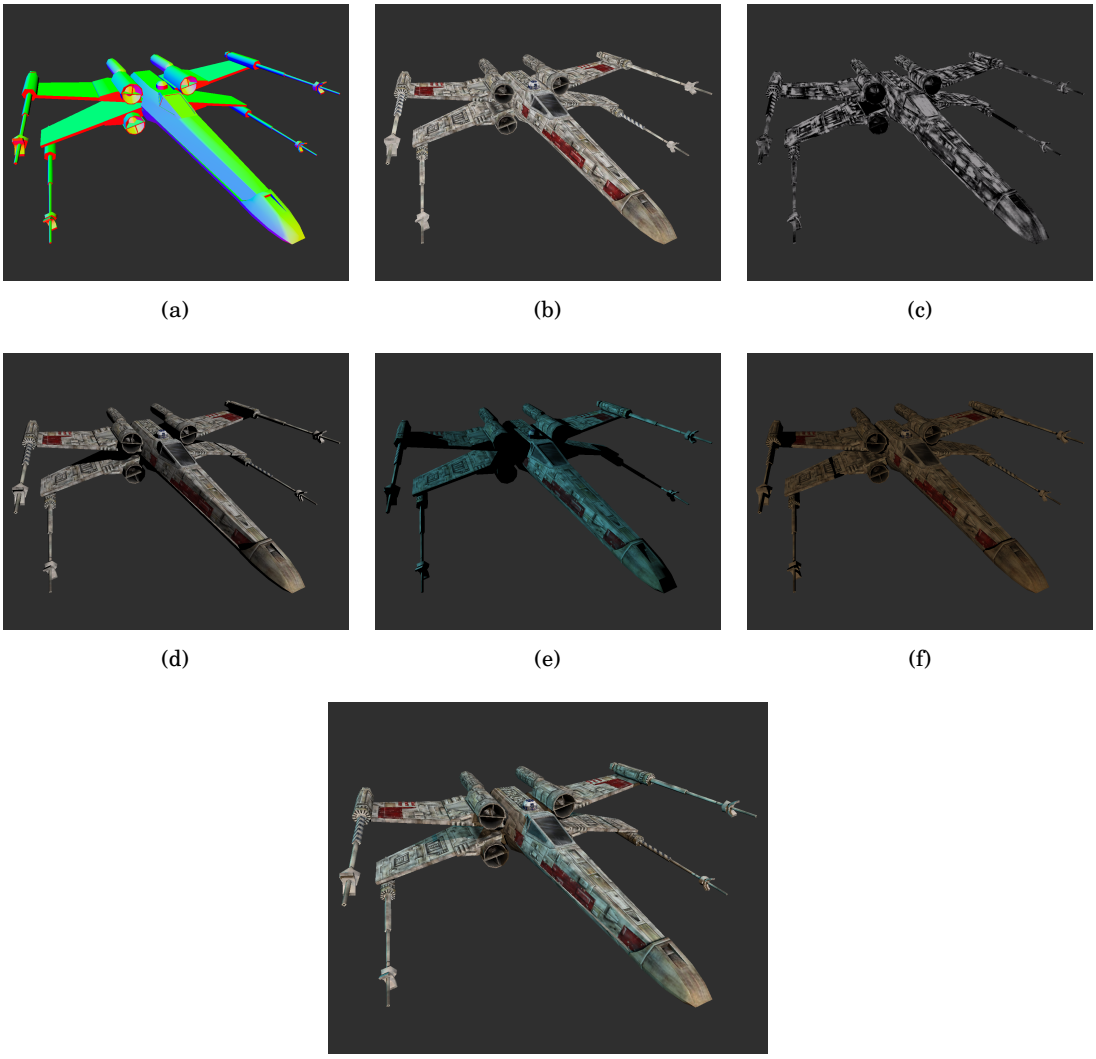
Figure 4: A local view of how a point **p** on the surface of an object is shaded by a single light source. The part of light coming from the direction **l** that is reflected towards the eye in direction **v** largely depends on the incident angle θ. The normal vector **v** denotes the orientation of the surface relative to the light source and is central to most shading models. The angle α_r between the view vector **v** and the reflected light vector **r** is directly proportional to the specular portion of reflected light (e.g., shiny highlights or direct reflection), whereas the diffuse portion of reflected light only depends on θ.

corresponds the the portion of the incident light that is mostly reflected into the direction reflected around the surface normal, which we denote as **r**. This models effects such as shiny highlights and mirror-like effects. In the Phong model, the specular component c_s that is reflected in the direction **v** is estimated based on the angle α_r between **v** and **r**:

$$c_s = \begin{cases} k_s \cdot (\max(\cos\alpha_r, 0))^m & \text{if } \cos\theta > 0 \\ 0 & \text{else} \end{cases} \tag{4}$$

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.

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 Between Ray and Plane

With plane we understand an indefinitely extending two-dimensional surface. In vector notation we can define a plane with $\langle \mathbf{p} - \mathbf{p}_0, \mathbf{n} \rangle = 0$, where $\mathbf{n}$ is the normal vector, $\mathbf{p}$ a set of points and $\mathbf{p}_0$ a point on the plane. Therefore we can describe a plane using a point and a normal vector, with $\mathbf{n}$ defining the orientation of the plane.

A ray $\mathbf{r}(t)$ is, as already mentioned, defined by $\mathbf{r}(t) = \mathbf{p} + \mathbf{d}t$, where $\mathbf{p}$ is the starting position and $\mathbf{d}$ the direction of the ray.

If $\langle \mathbf{d}, \mathbf{n} \rangle \neq 0$, then the ray intersects the plane in exactly one point. If $\langle \mathbf{d}, \mathbf{n} \rangle = 0$, the ray is parallel to the plane and no relevant intersection point exists. If an intersection point with a plane exists, we only need to find the corresponding ray parameter t.

Inserting into the plane equation we get:

$$\langle (\mathbf{p} + \mathbf{d}t) - \mathbf{p}_0, \mathbf{n} \rangle = 0, \tag{5}$$

and expanding gives

$$\langle \mathbf{d}, \mathbf{n} \rangle t + \langle \mathbf{p} - \mathbf{p}_0, \mathbf{n} \rangle = 0. \tag{6}$$

In the end we can solve for t and get

$$t = \frac{\langle \mathbf{p}_0 - \mathbf{p}, \mathbf{n} \rangle}{\langle \mathbf{d}, \mathbf{n} \rangle}. \tag{7}$$

In the framework a plane is defined by the normal $\mathbf{n}$ and an offset w in the direction of the normal $\mathbf{n}$.

The point on the plane $\mathbf{p}_0$ can be replaced by w and we get:

$$t = \frac{w - \langle \mathbf{p}, \mathbf{n} \rangle}{\langle \mathbf{d}, \mathbf{n} \rangle}. \tag{8}$$

5 Triangle Meshes

Triangle meshes are a universal approach to represent object surfaces. Any arbitrarily shaped surface can be approximated with arbitrary precision via a triangle mesh, where better approximation requires more triangles and thus, more resources. In order to render an object that is represented via a triangle mesh with ray tracing, we need to determine the intersection point of any given ray with any given triangle.

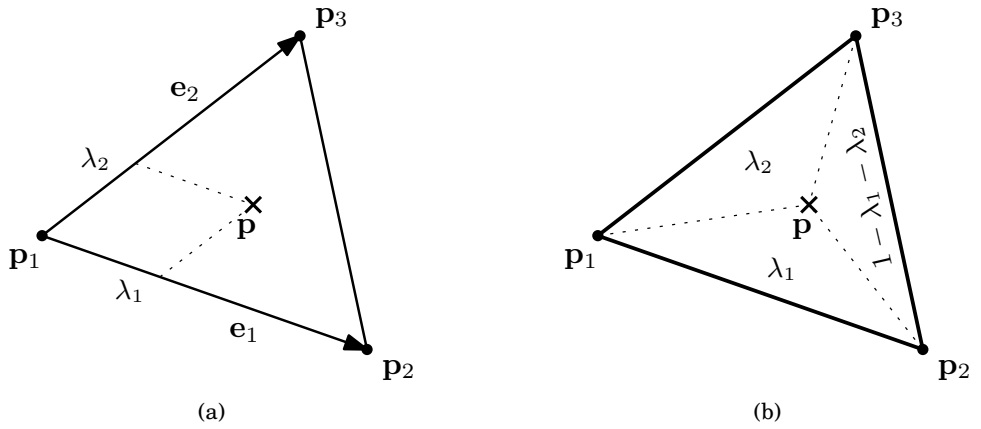


Figure 6: (a) A point $\mathbf{p}$ on the inside of a triangle can be represented as a linear combination of two edge vectors relative to a vertex. The parameters λ_1 and λ_2 correspond to the coordinates of the point $\mathbf{p}$ in the coordinate system that is defined by the edge vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ with the origin at $\mathbf{p}_1$. (b) Similarly, the point $\mathbf{p}$ can also be represented as a weighted sum of the vertices $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_3$, where λ_1 and λ_2 correspond to the area of the subtriangles that are formed with the point $\mathbf{p}$.

Any point $\mathbf{p}$ on the plane that is formed by the triangle vertices $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_3$ can be defined as a linear combination of two edge vectors relative to the shared vertex of these edges (see Figure 6a):

$$\mathbf{p}(\lambda_1, \lambda_2) = \mathbf{p}_1 + \lambda_1 \mathbf{e}_1 + \lambda_2 \mathbf{e}_2, \tag{9}$$

λ_1 and λ_2 denote the coordinates of the point $\mathbf{p}$ in the coordinate system defined by $\mathbf{e}_1$ and $\mathbf{e}_2$ with its origin in $\mathbf{p}_1$.

With the definition of the edge vectors

$$\mathbf{e}_1 = \mathbf{p}_2 - \mathbf{p}_1 \qquad\qquad \mathbf{e}_2 = \mathbf{p}_3 - \mathbf{p}_1$$

we can derive the equivalent form (using Equation (9))

$$\mathbf{p}(\lambda_1, \lambda_2) = (1 - \lambda_1 - \lambda_2) \cdot \mathbf{p}_1 + \lambda_1 \cdot \mathbf{p}_2 + \lambda_2 \cdot \mathbf{p}_3. \tag{10}$$

The weights λ_1 and λ_2 are commonly referred to as *barycentric coordinates*[1] of the point $\mathbf{p}$ (see Figure 6b).

The intersection point of a ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ with a triangle can then by derived by starting from the fact that an intersection point must lie on the plane of the triangle:

$$\mathbf{r}(t) = \mathbf{p}(\lambda_1, \lambda_2) \quad \Leftrightarrow \quad \mathbf{o} + t\mathbf{d} = \mathbf{p}_1 + \lambda_1 \mathbf{e}_1 + \lambda_2 \mathbf{e}_2.$$

[1]The name "barycentric coordinates" derives from the fact that the weights λ_1, λ_2 and $1 - \lambda_1 - \lambda_2$ directly correspond to the mass that you would need to attach at each vertex to move the triangle centroid to the point $\mathbf{p}$.

Reordering the equation, we arrive at

$$-t\mathbf{d} + \lambda_1\mathbf{e}_1 + \lambda_2\mathbf{e}_2 = \mathbf{o} - \mathbf{p}_1. \tag{11}$$

This vector equation is a system of three linear equations with three unknowns t, λ_1 and λ_2. We can introduce the matrix $\mathbf{A} = \begin{pmatrix}-\mathbf{d} & \mathbf{e}_1 & \mathbf{e}_2\end{pmatrix}$ and the vector $\mathbf{q} = \mathbf{o} - \mathbf{p}_1$ to rewrite the system of equations to:

$$\mathbf{A} \cdot \begin{pmatrix} t \\ \lambda_1 \\ \lambda_2 \end{pmatrix} = \mathbf{q}.$$

If the vectors $-\mathbf{d}, \mathbf{e}_1, \mathbf{e}_2$ are linearly independent, this matrix is invertible.
The solution of this system is

$$\begin{pmatrix} t \\ \lambda_1 \\ \lambda_2 \end{pmatrix} = \mathbf{A}^{-1} \cdot \mathbf{q}. \tag{12}$$

For a 3×3 matrix $\mathbf{M} = \begin{pmatrix}\mathbf{m}_1 & \mathbf{m}_1 & \mathbf{m}_2\end{pmatrix}$ the inverse can be computed as

$$\mathbf{M}^{-1} = \frac{1}{\langle \mathbf{m}_1, \mathbf{m}_2 \times \mathbf{m}_3 \rangle} \begin{pmatrix} (\mathbf{m}_2 \times \mathbf{m}_3)^T \\ (\mathbf{m}_3 \times \mathbf{m}_1)^T \\ (\mathbf{m}_1 \times \mathbf{m}_2)^T \end{pmatrix}. \tag{13}$$

For $\mathbf{A}$, we thus arrive at

$$\mathbf{A}^{-1} = \frac{1}{\langle -\mathbf{d}, \mathbf{e}_1 \times \mathbf{e}_2 \rangle} \begin{pmatrix} (\mathbf{e}_1 \times \mathbf{e}_2)^T \\ (\mathbf{e}_2 \times (-\mathbf{d}))^T \\ (-\mathbf{d} \times \mathbf{e}_1)^T \end{pmatrix} = \frac{1}{\langle -\mathbf{d}, \mathbf{e}_1 \times \mathbf{e}_2 \rangle} \begin{pmatrix} (\mathbf{e}_1 \times \mathbf{e}_2)^T \\ (\mathbf{d} \times \mathbf{e}_2)^T \\ (\mathbf{e}_1 \times \mathbf{d})^T \end{pmatrix}.$$

If we insert this into Equation (12), we arrive at

$$\begin{pmatrix} t \\ \lambda_1 \\ \lambda_2 \end{pmatrix} = \frac{1}{\langle -\mathbf{d}, \mathbf{e}_1 \times \mathbf{e}_2 \rangle} \begin{pmatrix} (\mathbf{e}_1 \times \mathbf{e}_2)^T \\ (\mathbf{d} \times \mathbf{e}_2)^T \\ (\mathbf{e}_1 \times \mathbf{d})^T \end{pmatrix} \cdot \mathbf{q} = \frac{1}{\langle -\mathbf{d}, \mathbf{e}_1 \times \mathbf{e}_2 \rangle} \begin{pmatrix} \langle \mathbf{e}_1 \times \mathbf{e}_2, \mathbf{q} \rangle \\ \langle \mathbf{d} \times \mathbf{e}_2, \mathbf{q} \rangle \\ \langle \mathbf{e}_1 \times \mathbf{d}, \mathbf{q} \rangle \end{pmatrix}.$$

By using $\langle \mathbf{a}, \mathbf{b} \times \mathbf{c} \rangle = \langle \mathbf{b}, \mathbf{c} \times \mathbf{a} \rangle = \langle \mathbf{c}, \mathbf{a} \times \mathbf{b} \rangle$, we can further simplify to

$$\begin{aligned} \langle -\mathbf{d}, \mathbf{e}_1 \times \mathbf{e}_2 \rangle &= \langle \mathbf{e}_1, \mathbf{e}_2 \times (-\mathbf{d}) \rangle = \langle \mathbf{d} \times \mathbf{e}_2, \mathbf{e}_1 \rangle, \\ \langle \mathbf{e}_1 \times \mathbf{e}_2, \mathbf{q} \rangle &= \langle \mathbf{q} \times \mathbf{e}_1, \mathbf{e}_2 \rangle, \\ \langle \mathbf{e}_1 \times \mathbf{d}, \mathbf{q} \rangle &= \langle \mathbf{q} \times \mathbf{e}_1, \mathbf{d} \rangle \end{aligned}$$

and

$$\begin{pmatrix} t \\ \lambda_1 \\ \lambda_2 \end{pmatrix} = \frac{1}{\langle \mathbf{d} \times \mathbf{e}_2, \mathbf{e}_1 \rangle} \begin{pmatrix} \langle \mathbf{q} \times \mathbf{e}_1, \mathbf{e}_2 \rangle \\ \langle \mathbf{d} \times \mathbf{e}_2, \mathbf{q} \rangle \\ \langle \mathbf{q} \times \mathbf{e}_1, \mathbf{d} \rangle \end{pmatrix}. \tag{14}$$

Finally, we can determine the ray parameter t as well as the barycentric coordinates λ_1 and λ_2 of the intersection point between the ray and the plane of the triangle:

$$t = \frac{\langle \mathbf{q} \times \mathbf{e}_1, \mathbf{e}_2 \rangle}{\langle \mathbf{d} \times \mathbf{e}_2, \mathbf{e}_1 \rangle} \qquad \lambda_1 = \frac{\langle \mathbf{d} \times \mathbf{e}_2, \mathbf{q} \rangle}{\langle \mathbf{d} \times \mathbf{e}_2, \mathbf{e}_1 \rangle} \qquad \lambda_2 = \frac{\langle \mathbf{q} \times \mathbf{e}_1, \mathbf{d} \rangle}{\langle \mathbf{d} \times \mathbf{e}_2, \mathbf{e}_1 \rangle}. \tag{15}$$

As visible above, no intersection point exists if $\langle \mathbf{d} \times \mathbf{e}_2, \mathbf{e}_1 \rangle = 0$. If an intersection point exists, the barycentric coordinates can be used to determine if the intersection point is inside the triangle. The intersection point is inside the triangle iff

$$\lambda_1 \geq 0 \quad \wedge \quad \lambda_2 \geq 0 \quad \wedge \quad \lambda_1 + \lambda_2 \leq 1. \tag{16}$$

6 Assignment Tasks

The goal of this exercise is to develop a simple ray tracing system with an arbitrarily oriented camera and phong shading. Additionally, triangle meshes should be supported, which requires the ray-triangle intersection discussed in the previous section. 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, as well as rays and cones. This will allow you to test components individually even if your triangle 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.

For submission 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)
```

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. For intersection tests with triangles, you will first need to implement `findClosestHitTriangles()` (see Section 6.4, same for planes). 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 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-Plane Intersection (2 Points)

Implement the function

```
const Plane* findClosestHitPlanes(const float3& p,
   const float3& d,
   const Plane* planes,
   std::size_t num_planes,
   float& t)
```

in «task1.cpp», so that for a given ray you calculate the nearest intersection with a plane in the given set of planes. The parameters p and d are the starting point and the direction of the ray. The set of planes is given as an array of planes with num_planes elements of type Plane. Every Plane contains the necessary plane parameters for the intersection computation in plane->p. If an intersection point has be found, set t to the corresponding ray parameter and return a pointer to the **nearest** hit plane. If no intersection point exists, return a null-pointer.

To get the intersection point of a single plane, call and implement the function

```
bool intersectRayPlane(const float3& p,
    const float3& d,
    const float4& plane,
    float& t)
```

as described in section 4. The parameters p and d once again contain the starting point and direction of the intersecting ray. The plane parameters **n** and the offset w are given by plane, where **n** is defined by plane.x, plane.y and plane.z and w by plane.w. If an intersection has been found, set t to the corresponding ray parameter and return true. If no intersection point exists, the function should return false.

6.4 Ray-Triangle Intersection (4 Points)

Here, your task is to implement

```
const triangle_t* findClosestHitTriangle(
      const float3& p,
      const float3& d,
      const triangle_t* triangles,
      std::size_t num_triangles,
      const float3* vertices,
      float& t,
      float& lambda_1, float& lambda_2)
```

in «task1.cpp». This function should find the *closest* intersection point between the given ray and the given triangles. The arguments p and d contain the ray origin and the ray direction as defined in Section 1. The triangles are given as an array triangles with num_triangles of the data type triangle_t. Each triangle_t is again an array of the three indices of a triangle that are used to index into the vertices array and load the vertices of the triangle. This means to load the second vertex (which has the index 1 due to 0-based indexing in C++) of the triangle with index 42, you would access it as vertices[triangles[42]][1]. This enables compact storage of triangles, and is a very common way to store triangles (indexed triangle list using an index buffer and a vertex buffer).

If an intersection point was found, the output parameters t, lambda_1 and lambda_2 should

be set to the correct ray parameter and barycentric coordinates of the intersection point (see Section 6.4). Additionally, if an intersection point was found, the function should return a pointer to the intersected triangle. If no intersection point exists, the function should return `nullptr`.

To get the intersection point of a single triangle, call and implement the function

```
bool intersectRayTriangle(const float3 &p, const float3 &d,
const float3 &p1, const float3 &p2, const float3 &p3,
float &t, float &lambda_1, float &lambda_2)
```

as described in section 5. The parameters `p` and `d` once again contain the starting point and direction of the intersecting ray. The triangle vertices $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_3$ are given by parameters of the same names. If an intersection has been found, set `lambda_1` and `lambda_2` to the corresponding barycentric coordinates and `t` to the corresponding ray parameter and return `true`. If no intersection point exists, the function should return `false`.

6.5 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 $\mathbf{o}$ 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.

6.6 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:

$$\mathbf{r}(t) = \mathbf{p} + t \cdot \mathbf{d} \qquad \text{subject to} \qquad t \in [t_{min},\ t_{max}]. \tag{17}$$

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 7. 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` 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.

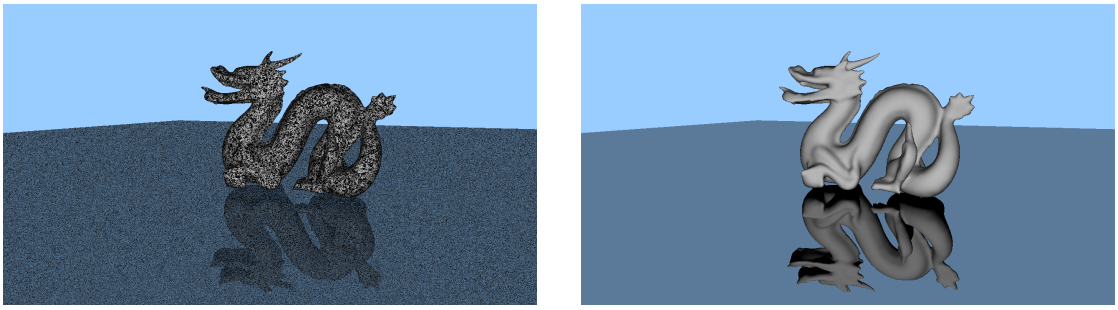
The intersection test between rays and triangles has to be implemented first in

```
bool intersectsRayPlane(const float3& p,
    const float3& d,
    const Plane* planes,
    std::size_t num_planes,
    float t_min,
    float t_max)
```

and

```
bool intersectsRayTriangle(
     const float3& p, const float3& d,
     const triangle_t* triangles,
     std::size_t num_triangles,
     const float3* vertices,
     float t_min, float t_max)
```

This function should return `true` if the given ray with origin `p` and direction `d` intersects *any* of the given triangles or planes, otherwise it should return `false`. Important to note is that you must only consider intersection points between `t_min` and `t_max` on the given ray.



(a) (b)

Figure 7: (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.

6.7 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->`$= \infty$)[2] are treated as mirror surfaces. For such surfaces, you should compute the reflected direction by reflecting the incident light direction $\mathbf{l}$ across the surface normal $\mathbf{n}$, 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 does not hit a non-reflective surface, you should assign it the `background_color`.

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