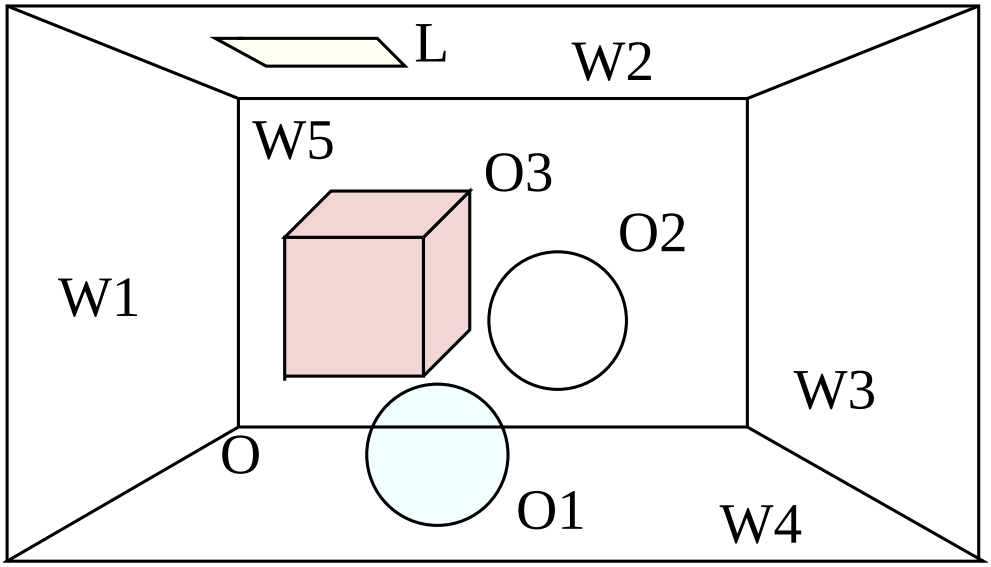
The global illumination renderer
Proposed class structure

An overview for a class structure of your global illumination renderer is proposed. It is recommended that you implement it using an object-oriented programming language such as Java or C++.

The scene

The scene should look approximately like this: The room is bounded by

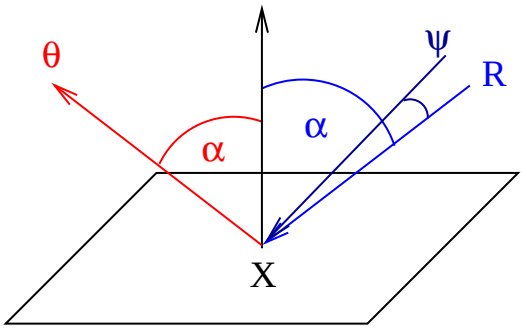


the walls *W1*-*W5*, which are defined by an object of the class *Wall*. The front side is open. The room coordinate system is the world coordinate system. The origin *O* could be the corner at the bottom left of the wall *W5*. The walls should be Lambertian (perfect diffuse) reflectors.

Three objects *O1*-*O3* should be inside the room. You should have at least one object of the class *Sphere* and one of the class *Cube*. The third object can be an object of the class *Sphere* or of the class *Cube*. At least one of the objects should be transparent. Transparent objects reflect light in both ray-tracing methods according to the perfect reflection / refraction law. The intransparent object should be a perfect reflector (mirror) in the Whitted

ray-tracer. Phong's reflection model with exponent n should be used for the BRDF in the Monte Carlo ray-tracer. The integral over the hemisphere with the perfect reflection direction $\mathbf{R}$ for the direction θ and the integration direction ψ (both in a cartesian representation) becomes:

$$L_r(x \to \theta) = \frac{(n+2)}{2\pi} \int_{\Omega_x} (\mathbf{R} \cdot \psi)^n L(x \leftarrow \psi) \cos(N_x, \psi)\, d\omega_\psi.$$



You should have one rectangular area light source (Lambertian emitter with a constant L_e) in the Monte Carlo ray-tracer. The area light source (L in the scene plot) should be on the ceiling $W2$.

You should have two point light sources on two different walls in the Whitted ray-tracer. The light sources can be integrated into the method of the *Ray* class that calculates the contribution from the local lighting model.

The Camera class

The camera should be one object. The class should be defined as follows:

Instance variables:

Position: The eye position is specified in the world coordinate system.

View direction: This unit vector should start at the eye position and point orthogonally to the view plane. The ray defined by the view direction should intersect the view plane at its central point.

View plane distance: The distance along the view direction between the view plane and the eye position.

View plane resolution: You should set a fixed size of the view plane, for example over the interval $[-0.5, 0.5] \times [-0.5, 0.5]$, and subdivide the view plane into pixels. The number of pixels should be at least 512×512.

Rays per pixel: Specifies how many rays you want to trace for each pixel.

Pixels: You initialize an array of pointers to objects of the class *Pixel*.

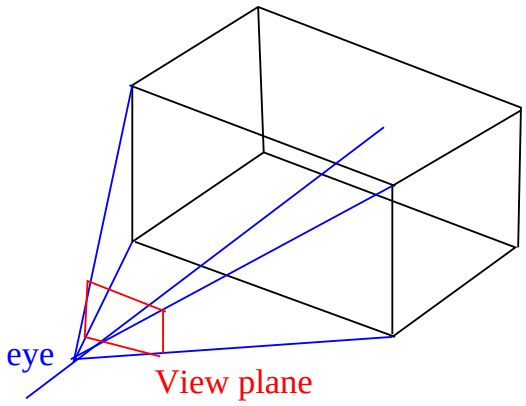
Instance methods:

Render image: Once this method is called, your renderer should compute the image according to the camera settings above. You have to execute a loop over all pixels. You can execute this loop on multiple cores by using OpenMP on a multi-core CPU.

Mapping function: This method should convert your radiometric pixel colour value into a photometric one.

Display image: Calling this method should plot the figure on the screen or save the image data onto disk if you want to display the rendered scene with other software.

How to set up the camera: The view direction is orthogonal to the wall W5 and it intersects the wall at its midpoint. This confines all possible eye positions to one straight line. Now you can set the eye's position and the distance between the eye and the view plane as follows: the rays from the eye to the 4 corner points of the room on the side, which is facing the camera, should intersect the 4 corners of the view plane. The camera captures in this case the entire room. You can change the camera's perspective by moving the eye point and by changing the distance from the eye to the view plane while you keep the view plane size unchanged.



The Pixel class

Each pixel of the *Camera* object should correspond to one instance of the class *Pixel*. It should have the following instance variables and methods.

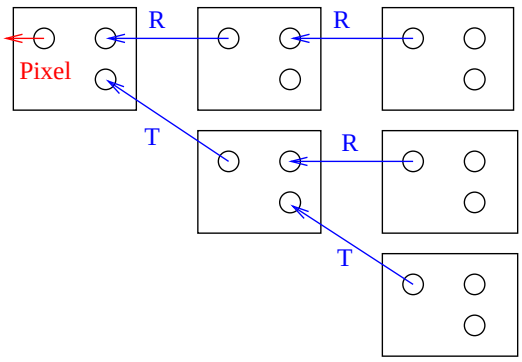
Instance variables:

Array of rays: You should have an array that holds the addresses of all rays, which are launched at the eye point and go through this pixel.

Colour of pixel: An equal weight is given to the contributions of all the rays, which were launched at this pixel, to the final colour.

Instance methods:

Shooting rays: You generate points on the pixel with a random number generator. The vectors from the eye point to the random points correspond to the ray directions. The rays are stored in the array of rays of the *Pixel* object. The number of rays is determined by the instance variable *Rays per pixel* of the *Camera* object and it should be the same for all pixels. The method *Shooting rays* should track the ray until its end. Hence you build up the tree structure, which we have discussed in the lecture 2. You develop a linked list for all rays that are spawned by the initial ray. Each block



corresponds to a *Ray* object and the one to the left is attached to the *Pixel* object. Each *Ray* object has pointers to children rays (blue rays). Each *Ray* can have multiple children. The sketch exemplifies this for a scene that is ray-traced by Whitted's scheme and where you have reflected rays R and refracted rays T. Although you can also spawn multiple rays in the Monte-Carlo scheme, it is usually better to have only one reflected ray at each ray-surface intersection and to launch more rays at the camera.

No children *Ray* should be spawned in the Whitted ray-tracer or in the Monte Carlo ray-tracer when a *Ray* hits the *Wall* object or when it leaves the scene to the side facing the camera. Russian roulette should limit the number of rays in the Monte Carlo ray-tracer. The maximum number of ray iterations in the Whitted ray-tracer should be set to 10.

The Ray class

An instance of the class *Ray* corresponds to a node in the tree structure of the ray-tracer (Lecture 2).

Instance variables

Starting point: This is the eye point for a ray that is launched through a pixel. The point of intersection between a ray and a surface is the starting point for the next ray along the linked list.

Direction: This vector should have unit length. The direction of the first ray is defined by the eye point and by the point on the pixel, which was computed with a random number generator. The direction of a ray, which is created by a reflection or refraction, is determined as follows: It is the perfect reflection direction or the perfect refraction direction for transparent objects. It is the perfect reflection direction for intransparent objects in Whitted ray-tracing. It is computed by a random number generator for random azimuth and inclination angles for Monte Carlo ray-tracing.

Importance: This variable has the information about how much importance the ray got from the parent ray. The importance of the first ray is $1/N_{pixel}$, where N_{pixel} is the number of rays that go through that pixel. This number is defined by *Rays per pixel* in the *Camera* class. The importance of the other rays is computed for transparent objects via the perfect reflection / refraction laws. It equals the importance of the parent ray for an intransparent object in the Whitted ray-tracing scheme and it is determined by the BRDF for Monte Carlo ray-tracing of intransparent surfaces.

Colour: This is an RGB vector that determines the colour for the ray under consideration.

Final node: You can use a Boolean to mark a ray that has hit a wall, left the room or that is the last one in the branch of the tree.

Pointers to children nodes: The number of children nodes depends on the numerical scheme. Whitted ray-tracing gives you one child ray for in-

transparent objects and two children rays for transparent ones. Transparent objects give two children rays for Monte Carlo ray-tracing. Monte Carlo ray-tracing of intransparent objects may give many children rays. Choosing one child ray is the simplest case and it gives good results.

Inside/outside of the object: You can set a Boolean to *true* if the ray is inside an object and to *false* if it is not. This information can be used to determine if you have to pay attention to the critical incidence angle (Page 16, lecture 2).

Instance methods

Calculate colour: Here you combine the colour contributions from the children nodes and from a local lighting model to a single RGB vector. This RGB vector corresponds to the colour of the ray C_R. Assume that you have two children rays with the importances W_1 and W_2 and the colours C_1 and C_2 and that you calculate the colour C_3 from a local lighting model with the importance W_3. You evaluate the local lighting model at the position of the children nodes. The ray colour is $C_R = W_1C_1 + W_2C_2 + W_3C_3$ with $W_1 + W_2 + W_3 \leq 1$. The condition is $W_1 + W_2 + W_3 = 1$ for Whitted ray-tracing.

You calculate the importance with the help of the Monte Carlo approximation for the reflected radiance (See also page 2)

$$L_r(x,\theta) = \sum_{i=1}^{N} \left(\frac{n+2}{2\pi N} \frac{(\mathbf{R}\cdot\psi_i)^n \cos(N_x,\psi_i)}{p(\psi_i)} \right) L(x \leftarrow \psi_i),$$

where $p(\psi_i) = p(\theta_i, \rho_i)$ is the probability distribution function for the inclination angle θ and the azimuth angle ρ evaluated at the random numbers θ_i and ρ_i. The importance given by Phong's BRDF is the term in the parentheses. Note that the integral for the importance calculation is given in a hemispherical form. You need to use the integral over surfaces and get the equivalent Monte Carlo approximation (See Lecture 4).

Calculate local lighting contribution: You compute $L_e(x,\theta)$ for the point where the ray intersects a surface. In the case of Whitted ray tracing, you shoot shadow rays from the ray-surface intersection point to the point light sources (you can store their positions in this instance method). You test for visibility. If the light sources are visible, you calculate their contribution to the L_e-term using Phong's shading model. Assume that shadow rays can go through transparent objects without being reflected or refracted.

The contribution from the local lighting model in the Monte Carlo ray-tracer works as follows. Random positions on the area light source are computed and these positions are connected to the surface point under consideration (Lecture 5). You use Phong's shading model to evaluate the radiance contribution to the outgoing ray.

Note that a local lighting component based on Phong shading is only applied to the transparent and intransparent objects in the scene, while the walls are perfect diffuse (Lambertian) reflectors.

The Sphere class

This class contains information about an object and its location in the scene and it allows you to compute ray-surface intersections.

Instance variables

Position: This is the position at which the *Sphere* is centred.

Radius: The radius of the *Sphere*.

Transparency: A boolean that is true if the object is transparent.

Refractive index: This number determines the refractive index of the *Sphere*. Air has an index ≈ 1.0 while glass has an index ≈ 1.5. You should select a value in this range.

Instance methods

Calculate intersection: This method calculates the intersection points between a ray and the surface of the *Sphere*. You equate the equation for the ray to the equation of the *Sphere* and calculate from it the intersection points. Zero intersections corresponds to a miss. One intersection corresponds to a tangential hit. If you get two intersection points and the ray is located outside of the sphere, then you determine the correct intersection point by comparing the length of the vectors from the ray's origin to the two intersection points. Select the intersection point that is closer to the *Ray*s starting point. If you are inside an object, then you take the intersection point that is not the *Ray*s starting point. Make sure that the intersection points are not located inside the object because of round-off errors. If they are, then you should move them along the surface normal until they are just outside of it.

Calculate children rays: The number of children rays depends on the surface material of individual objects in the scene, so we should make it an instance method of the objects. The new rays are appended at the appropri-

ate position in the tree data structure of the rays.

The Rectangle class

You have to implement *Cube* objects, *Wall* objects and *Light* objects in your scene. It is thus beneficial to define a *Rectangle* class that is used by all these objects.

Instance variables

Positions of the corner points: The positions of the corner points are initialized from the *Cube*, *Wall* or *Light* objects.

Instance methods

Calculate intersection: You equate the ray equation to the equation of a plane and you get the coordinates of the intersection point between the *Ray* and the *Rectangle* if it exists.

Computation of children rays: This method is only called if the ray intersects a *Cube*. A ray, which hits a *Wall* object or a *Light* object, does not have children. The intersection of a *Ray* with a *Wall* or *Light* will contribute to the radiance of the ray via the L_e term of the rendering equation.

The Cube class

A *Cube* object contains an array with pointers to 6 *Rectangle* objects and methods that computes the children rays.

Instance variables

Position: It defines the position of one corner of the *Cube*.

Size: This scalar variable sets the length of the edges of the *Cube* object. All edges have the same length and only one value for *Size* is needed.

Transparency: This boolean is set to true if the *Cube* is transparent.

Refractive index: If the *Cube* object is transparent then the refractive index should be between 1 and 1.5.

Instance methods

Initialization of the Rectangle-objects: You can choose the corner points of a *Cube* object in its local system and combine them to *Rectangle*s such, that the length of each edge is 1. We assume that the edges of the *Cube*

object are parallel to the axes of the world coordinate system. You multiply the vectors, which point from the origin of the *Cube*'s local coordinate system to the corner points of the rectangles, by *Size* and add to it *Position*. These steps transform the coordinates of the *Cube* into the world coordinate system.

Computation of the children rays: You make a loop over all *Rectangle* objects of the *Cube* object and you compute the intersection of the ray with each *Rectangle*. If the ray intersected the *Cube* object for the first time, then you pick the intersection point that is closest to the base of the ray. If the ray is inside the *Cube* object then you take the intersection between the ray and the *Cube* object that is not the *Ray*s starting point. The number of child rays depends again on the transparency (and if you have transmitted and reflected rays), on whether you use Whitted ray-tracing or Monte Carlo ray-tracing, and on how many rays you use in the Monte Carlo scheme. You compute the importances of the children rays using the perfect reflection and/or refraction laws or the BRDF (Phong model) of the intransparent surface.

The Wall class

A wall object contains an array with pointers to 5 *Rectangle* objects.

Instance variables

Position: It defines the position of one corner of the *Wall* object. You can set it to (0,0,0) and define it as the origin **O** (See figure on page 1).

Size: This scalar variable sets the length of the edges of the *Wall* object. All edges have the same length and only one value for *Size* is needed.

Instance methods

Initialization of the Rectangle objects: The positions of the corners of each *Rectangle* are defined in its local coordinate system and the edge length is 1. For simplicity we assume that the edges of the *Rectangle* objects of the *Wall* are parallel to the axes of the world coordinate system. You can thus initialize the corners of each of the 5 *Rectangle*'s assuming that its surface orientation is orthogonal to one of the axes of the world coordinate system. Then you multiply the vectors by *Size* and add to it *Position*.

Computation of intersections: You compute the intersection between the *Ray* and the *Wall* object by running a loop over all its *Rectangle* objects. You can have either one intersection or none (the ray leaves through the front of the room). A *Ray* that hits a wall does not spawn children *Ray*s. You

only calculate the L_e term at the intersection, assuming that your wall is a Lambertian reflector.

The Light class

A light source holds a pointer to a *Rectangle* object and provides additional functionality. It should be placed approximately in the center of the ceiling (see figure on page 1). The light source plane should be contained in the plane of the ceiling. If a *Ray* hits a light source directly, then you do not use a local lighting model to evaluate L_e. You feed the light source's L_e directly into the *Ray*.

Instance variables

Radiance: You set a value for L_e, which corresponds to the emitted radiance. The light source should be a perfectly diffuse emitter.

Position: The position of the light source's *Rectangle*.

Size: The size of the light source's *Rectangle*. Its side length should be about 1/6 times that of the ceiling's *Rectangle*.

Instance methods

Random position: This method should return a random position on the light source surface. This position is used together with a point on a surface to determine the direction of shadow rays.

Suggestions of what you can do in the results section

You should apply your global illumination renderer and present the results in your report. You can study various aspects, for example:

- Parametric study of refraction: You can show the same scene but with a series of studies where you vary the refractive index of your transparent object('s). You can show plots that use a refractive index of 1.1, 1.3 and 1.5 and compare them.

- Changing the resolution and the ray number such that the computing time stays the same: You can for example use a screen resolution of 512×512 pixels and 16 rays per pixel and one with 1024×1024 pixels

and 4 rays per pixel. You can then look at what gives you better pictures.

- You can compare renderings with many rays per pixel and one new ray per reflection with renderings that use few rays per pixel and more than one ray per reflection (For the Monte Carlo scheme).

- Varying the spectral index of the Phong model. You can compare scenes where your Phong model has a high value for n (the exponent of the scalar product) with scenes with a low value of n.

- Replacing the Fresnel equations of reflection / refraction with the small angle approximation (Lecture 2): You can examine how the transparent object changes when you go from the Fresnel equations for the reflected and refracted radiances to the small angle-approximation.

- Multi-core performace: You can check how well the code scales with the number of cores you use. The speedup should be linear.