1 Structure of the program

The program is divided into three main parts: The kd tree, raycasting and input file parsing.

2 Kd tree

The implemented kd tree is a three dimensional data structure that recursively subdivides the space into three dimensional rectangles each corresponding to some node in the tree. The leaves of the tree can then be used to store triagles. The kd tree can then be used to answer queries of the form "find the first triangle intersecting a ray r".

2.1 Kd tree construction

The algorithm used to build the Kd tree was the $\mathcal{O}(n \log^2 n)$ algorithm described in Wald et al., (2006) `https://ieeexplore.ieee.org/document/4061547`.

The algorihm works by recursively splitting the triangles into two subtrees according to the surface area heuristic.

The idea behind the surface area heuristic is to minimize the expected cost of the tree queries when the query rays are distributed randomly.

Given that a random ray hits a voxel V, then the probability of this ray hitting a subvoxel (of V) S is $SA(S)/SA(V)$ where $SA(x)$ is the surface area of voxel x.

The expected cost of a query can then be estimated if the cost of tree traveral K_t and intersection K_i are known.

The surface area heuristic estimates the goodness of a split s by assuming that the subtrees would be leaves. In this case the cost of node x would be

$$C_x(s) = K_t + K_i(\frac{SA(l)}{SA(x)}N(l) + \frac{SA(r)}{SA(x)}N(r))$$

Where l and r are the left and right nodes of x and $N(y)$ is the number of triangles in node y. $SA(y)$ is interpreted as the surface area of the voxel corresponding to node y.

As far as I understood , the K_t term is intentionally ignored for the leaf nodes.

For each node, the algorithm, considering every dimension separately, uses a sweeping line technique to evaluate $C_x(s)$ for every interesting s.

The algorithm then terminates when the estimate for the new split would be larger than simply setting current node to a leaf.

The complexity proof (in Wald et al., 2006) assumes that the splits will be quite good also in the terms of how many triangles are sorted into left and right subtrees. I don't know how to prove any bounds related to this.

Another issue in the time complexity is that a single triangle can be part of many subtrees. This requires to have a representation of a shape generated by intersecting triangle with half planes. In the program it was done by representing triangles as polygons and intersecting these with half-planes.

By assuming that at maximum $\mathcal{O}(\sqrt{N(x)})$ triangles will intersect with the splitting plane of node x, it can be shown that the algorithm runs in $\mathcal{O}(N \log^2 N)$.

2.2 Queries

Finding the first intersection is done by traversing the tree recursively always trying first the subtree closer to the origin of the ray. In case a subtree voxel doesn't intersect the ray, it can be skipped. When a leaf is reached, all of the triangles in that subtree are tested for intersection. Because the tree is traversed by first visiting the closer subtree, the algorithm can return the answer as soon as it find the closest intersecting triangle from any subtree.

The speed of these queries depends completely on the triangle set used to built the tree. In the worst case, all of the triangles have to be checked and the kd tree is not useful at all.

2.3 Quicksort

Quicksort was used to implement the sorting step in the kd tree building.

To ensure the $\mathcal{O}(n \log n)$ running time, the pivot was selected with the quick-select algorithm using median of medians. `https://en.wikipedia.org/wiki/Median_of_medians`.

Briefly, this is done by implementing two functions: `selectK(begin, end, k)` which returns the exactly k'th element of elements in the array range $[begin, end)$. The other function is `medianOfMedians(begin, end)` which will return an approximate median of elements in the range $[begin, end)$.

`medianOfMedians(begin, end)` works by splitting the array into chunks of five (possibly leaving the last chunk , forming an array from the medians of these chuncks, and calling `selectK` to find the (lower) median of the medians. The median of medians is guaranteed to be between the 3/10 and 7/10 quantiles of values on the range $[begin, end)$.

`selectK(begin, end, k)` works by selecting the pivot with `medianOfMedians`, partitioning the values according to the pivot, and then recursing to the elements containing the k'th elemnt.

If we now denote the running time of `selectK(begin, end, k)` by $T(N)$ where $N = \texttt{end} - \texttt{begin}$, we get the recursion:

$$T(N) \leq C \cdot N + T\left(\left\lfloor \frac{7}{10} N \right\rfloor\right) + T\left(\left\lceil \frac{1}{5} N \right\rceil\right)$$

Which can be proved to be linear by induction.

3 BSDF

BSDF (bidirectional scattering distribution function, the function only considering reflection is called BRDF [bidirectional reflectance distribution function]) is a function $f : R^3 \times S^2 \times S^2$ that describes how light travels when reflecting/refraction from a surface.

The radiance to direction ω_o from point p is given by

$$L_0(p, \omega_o) = \int_{S^2} f(p, \omega_i, \omega_o) L_i(p, \omega_i) |cos\theta_i| \, d\omega_i.$$

where $L_i(p, \omega_i)$ is the radiance coming to point p from the direction ω_i.

The *cos* term in the integral comes from the fact that radiance is defined for a plane perpendicular to the ray and that the radiance for a non-perpendicular surface will be smaller.

There are two desirable properties for a BSDF. First, it should be bidirectional meaning that the light ray should behave the same even if its direction is reversed. This property can be stated as $f(p, \omega_i, \omega_o) = f(p, \omega_o, \omega_i)$ for all pairs of ω_o and ω_i.

Unfortunately in the program this had to be relaxed in the case of specular reflection to handle the case of perfect reflections correctly (this was done simply by increasing a parameter in the BSDF).

Other property is that the BSDF should conserve energy: For every ω_i we require:

$$\int_{\Omega} f(\omega_i, \omega_o) \, |\cos\theta_o| \, , d\omega_o \leq 1$$

In the program this should hold for reflections and diffuse refractions. However, I was unable to reach a conclusion if this also holds for specular refraction implementation.

3.1 BSDF implementation

The way the program construct the final BSDF is by recursively taking the weighted mean of other, more specialized, BSDF functions.

In the following, let n be the normal of the ray intersection point that is facing to the same side of the plane as ω_o

All `opaque` functions get value 0 when $\omega_i \cdot n \leq 0$.

We define the the used BSDF as

$$f(p, \omega_i, \omega_o) = \texttt{diffuse} * f_{\texttt{diff}}(p, \omega_i, \omega_o) + \texttt{specular} * f_{\texttt{spec}}(p, \omega_i, \omega_o)$$

And require that $\texttt{diffuse} + \texttt{specular} \leq 1$. (Unless the user wants to violate the conservation of enery).

Further:

$$f_{diff} = \texttt{transparent} * f_{\texttt{diff\_trans}} + (1 - \texttt{transparent}) * f_{\texttt{diff\_opaque}}$$

And:

$$f_{specular} = \texttt{transparent} * f_{\texttt{spec\_trans}} + (1 - \texttt{transparent}) * f_{\texttt{spec\_opaque}}$$

Here `transparent` is meant to be interpreted as weight for behaviour where light can be both reflected and refracted. This is because a fully transparent object can both reflect and refract light according to the Fresnel equations. However, I wasn't sure if the Fresnel equations should affect the diffuse lighting so I just decided to handle them separately. In the context of diffuse lighting, a fully transparent object is an object that emits light with constant radiance to every direction:

$f_{\texttt{diff\_opaque}} = a/\pi$ where $a \leq 1$ for the conservation of energy.

$f_{\texttt{diff\_trans}} = a/(2\pi)$ because light will be distributed over twice as large area as in $f_{\texttt{diff\_opaque}}$.

3.1.1 Fresnel factor

For specular reflections, the Fresnel equation is used to find the correct weight for the reflected light. The intuitive idea behind this is that more light will be reflected when the light light hits the surface at very small angle relative to the surface.

This should also affect the non-transparent surfaces but this case was not taken into account in the program.

Reference: `https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel`

Calculating the Fresnel factor, the Schlick's approximation was used. `https://en.wikipedia.org/wiki/Schlick%27s_approximation`

For index of refraction we will assume that the objects are in vacuum so the `index_of_refraction` we get from the material file will be interpreted as the index of refraction on the side of the triangle that is opposite to the triangle normal.

3.1.2 Phong lighting model

For specular the Phong lighting model is used: `https://en.wikipedia.org/wiki/Phong_reflection_model`

The reason why the original Phong model instead of the modified (satisfying bidirectionality, `https://graphics.cs.kuleuven.be/publications/Phong/`), was used is that the modified Phong model couldn't handle the perfect reflections correctly.

Let r be the perfect reflection of ω_i. Let α be the angle between r and ω_o. Let θ_o be the angle between n and ω_o. Let θ_i be the angle between n and ω_i.

$$f_{\texttt{phong}} = \frac{(\texttt{specular\_exp} + 1)}{2\pi} \frac{(\cos\alpha)^{\texttt{specular\_exp}}}{\cos\theta_o}$$

The normalization factor is to ensure the conservation of energy. `http://www.thetenthplanet.de/archives/255`

The $1/\cos\theta_o$ term is needed to ensure that the perfect reflections (which are implemented by setting `specular_exp` $\approx 10^1 0$ or so) work correctly.

For perfect reflections, the output radiance of the perfectly reflected ray should be equal to the radiance of the input ray no matter the input ray's angle. However, in the formula

$$L_0(p,\omega_o) = \int_{S^2} f(p,\omega_o,\omega_i) L_i(p,\omega_i) |cos\theta_i| \, d\omega_i.$$

the input radiance is attenuated by the cosine term. For perfect reflections and for ω_i ω_o for which $f_{\texttt{phong}}(p,\omega_i,\omega_o) > 0$, the values of ω_i and ω_o are very close to each other. Therefore $1/\cos\theta_o$ from the $f_{\texttt{phong}}$ will cancel out the input randiance attenuation.

3.2 Specular opaque

Specular opaque will be simply:

$$f_{specular\_opaque} = f_{phong}$$

3.3 Specular transparent

Let θ_i be the angle between incoming ray and the plane normal.

Here we need to again split the BSDF. This time into three parts:

1. In case the θ_i is large enough, a total internal reflection will occur and f_{phong} can be used to handle this.
2. Otherwise the Fresnel factor $F(\theta_i)$ will be used to select between $f_{\texttt{specular\_opaque}}$ (reflection) and $f_{\texttt{specular\_refraction}}$.

Let's define $t(\theta_i) = [\theta_i \geq \texttt{critical\_angle}]$. Where `critical_angle` is the angle at which total reflection will start occurring. In other words $t(\theta_i)$ will be 1 if there is a total reflection and 0 otherwise. We get:

$$\begin{aligned} f_{\texttt{specular\_transparent}} = {} & t(\theta_i) f_{\texttt{phong}} \\ & + (1 - t(\theta_i)) \left(F(\theta_i) f_{\texttt{phong}} + (1 - F(\theta_i)) f_{\texttt{specular\_refraction}} \right) \end{aligned}$$

Let's denote by $r(x)$ the direction (unit vector) of refraction of x.

$$f_{specular\_refraction}(\omega_i,\omega_o) = f_{phong}(\omega_i, r(\omega_o))$$

Note that this is not necessarily the same as $f_{phong}(r(\omega_i),\omega_o)$ because the refraction can map the rays to a wider or narrower space on the other side of the plane.

4 Importance sampling

The integral

$$L_o(p,\omega_o) = \int_{S^2} f(p,\omega_i,\omega_o)L_i(p,\omega_i)|cos\theta_i|\,d\omega_i.$$

is evaluated by the program using the Monte Carlo integration:

$$L_o(p,\omega_o) \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(p,X_i,\omega_o)L_i(p,X_i)|\cos\theta_i|}{p(X_i)}$$

where X_i are random variables on a sphere with a probability density function p.

In other words, the program is estimating the color of p, seen from direction ω_o by shooting random rays from point p.

Two strategies are used by the program to sample the values X_i. First, the samples can be. First, the values can be sampled uniformly from the surface of a ball. This work to some extent with diffuse materials but fails very badly if the material has very tight specular reflections. This is because for the specular materials, the BSDF will return something very small for almost all directions except one and thus sampling those directions is somewhat meaningless.

To alleviate this, the samples can be drawn from a distribution more closely resembling the integrand. For this the program uses cosine exponent distribution with exponent `specular_exp` for specular lighting and with exponent 0 for diffuse lighting. (from here `https://graphics.cs.kuleuven.be/publications/Phong/`).

This method of sampling points from a distribution estimating the integrand is called importance sampling. The importance sampling could further be extended to better match the integrand by favouring the directions with emitting surfaces. The program does not do this and this results in somewhat poor performance if the only light source in the scene is a light emittins surface.

In addition, the contribution from point lights is added to L_o by going through them and, if the point light is visible, then applying the BSDF to its light.

As the BSDF is in the form

$$f(p,\omega_i,\omega_o)) = \sum_i c_i f_i(p,\omega_i,\omega_o)$$

the samples were taken from a distribution with pdf

$$pdf(p,\omega_i,\omega_o) = \sum_i c_i pdf_i(p,\omega_i,\omega_O)$$

where pdf_i is the importance sample distribution for BSDF i.

Reference: `http://www.pbr-book.org/3ed-2018/Monte_Carlo_Integration/Importance_Sampling.html`

5 Emission

In addition to reflection and refraction, the surfaces can also emit light in which case the emitted light is simply added to the surface points' outgoing radiance. This resembles a perfectly diffuse surface to which a light is constantly being shone to. (As a sidenote, this will lead to a phenomenon where the emitting surface will look equally bright when looked at any direction but will not illuminate surfaces equally to every direction.)

6 Raytracing

Raycasting is done recursively always to a certaini constant maximum recursion depth. At every intersection, a constant number of new rays is shot.

Pixels are sampled either once from their middle point of then from some number of random points. For the rendering, the pixels are split into some number of groups which are then processed by separate threads.