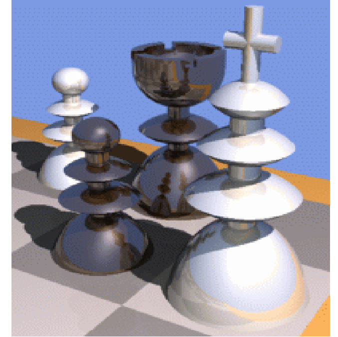
Ray-Tracing

Ray stands for a light ray and to trace means to follow a track. Thus, ray-tracing means to follow the tracks of light rays, however in reverse direction. Based on the principle idea of ray-casting, ray-tracing is a very powerful method, which – in addition to correct visibility - can simulate some important optical effects: shading, shadows, reflections, light refraction. Due to the simplicity of this method even very complex objects such as freeform surfaces, fractal surfaces or any mathematical function, can be rendered.

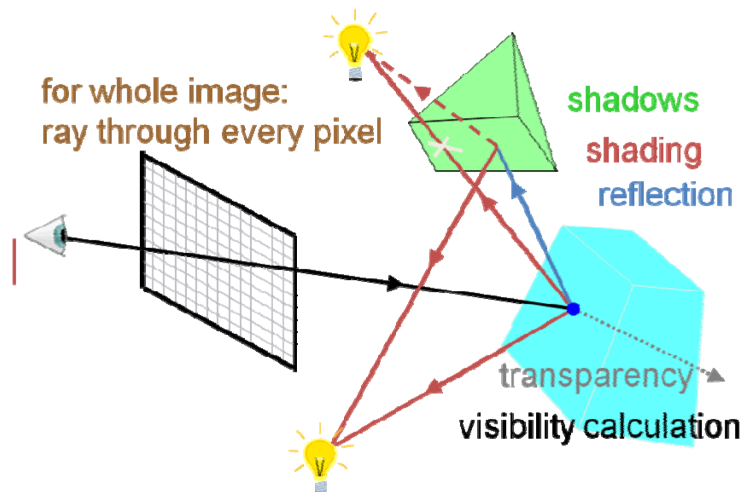


The Ray-Tracing Principle

The basic idea is to trace the ray of light that hits a pixel on the screen in reverse direction (i.e. to analyse its origin) and to infer the pixel's appearance from it.

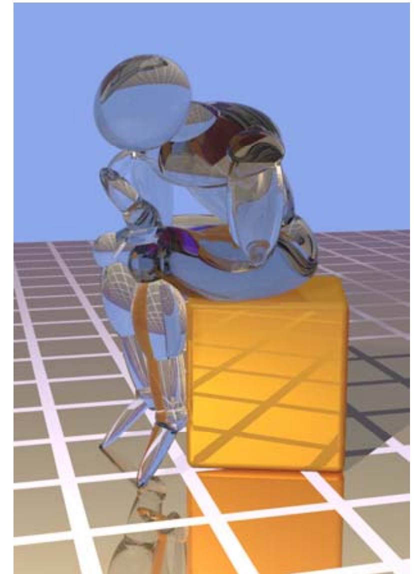
Correct Visibility and Shading
Like in ray-casting a viewing ray ("primary ray") is placed through every pixel and then intersected with *all* surfaces in the scene. From all the obtained intersection points the one closest to the screen is selected and the shading of that object point (as seen from the view point) determines the value for the pixel. Having done this for all points on the screen (i.e. in the simplest case for all pixels), we end up with a depiction of the scene in correct visibility. Any shading model can be used for this purpose, for example the Phong-model.

Shadows
In order to calculate a surface point's shading we not only need its surface normal but also the directions towards all light sources. A light source has direct influence on the shading of a point only if its light is not obscured by other objects, i.e. if there are no other objects between the point in question and the particular light source. To determine this, we cast a *shadow ray* (that is a "secondary ray") from the point to be shaded to the position of the light source. Then we intersect this straight line with *all* objects in the scene and ignore this light source if we find an intersection point between the object and the light source. In this way all object parts in the shadow of a light source blocking object obtain less influence from the occluded light source than object parts which are directly visible from the light source. A shadow is automatically created by the light-occluding objects.



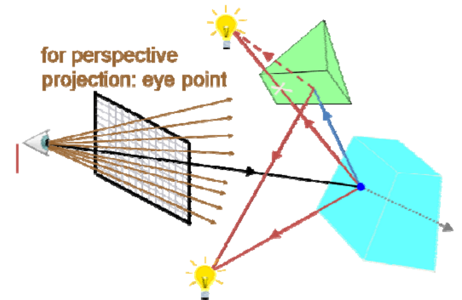
Reflections
If the viewing ray hits an object that is a mirror, the viewer doesn't see the object itself, but rather the object that is visible from the point of intersection along the direction of reflection. Since the law of reflection (incident angle equals emergent angle) is symmetric, the reflected object can be found by reflecting the viewing ray on the mirror-surface. Then we follow the *reflected ray* (also a "secondary ray"), i.e. again we intersect it with *all* objects in the scene and choose the closest intersection point. The new intersection point's shading (as seen from the surface point on the mirror) is now what the original viewing ray sees. Note that the reflection is calculated locally, which means that curved mirrors can be rendered without additional expenditure.

Transparency
Dealing with transparent objects is just as simple. If the first viewing ray's intersection point is on a transparent object, from the viewing direction we see what the *transparency ray* (again a "secondary ray") hits after having traveled through the transparent object. To simulate the correct behavior of light, it is not a problem to refract the ray at the transparent material according to the refraction law. The transparency ray is again intersected with *all* objects and the closest intersection point's shading (from the direction of the transparency ray's arrival) is assigned to what the original viewing ray sees.



Recursion
Except for shadow rays, every ray, i.e. every straight line representing a reversed light ray, is basically equal. For the action at that position it is irrelevant whether it is a primary or a secondary ray. In this way even multiple reflections and refractions behind transparent materials etc. are equally simple.

Perspective
The way primary rays are generated determines the projection of the scene to the view plane. When using parallel rays, orthogonal to the image plane, we obtain an orthonormal parallel projection. However, if we let the viewing rays emerge from a virtual eye point, we get a perspective projection. Note that in this way the perspective projection is generated naturally, without additional effort (apart from optimization procedures which take advantage of the parallel-ray-property).



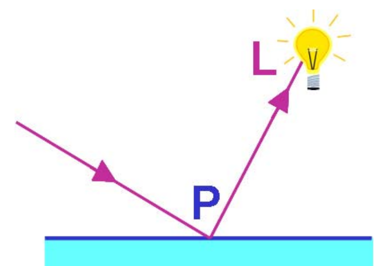
Ray-Tracing Implementation

Writing a ray tracer is fairly simple. All we need is a function that intersects a straight line with all objects and returns the closest intersection point.

Ray-tracing pseudo-code:

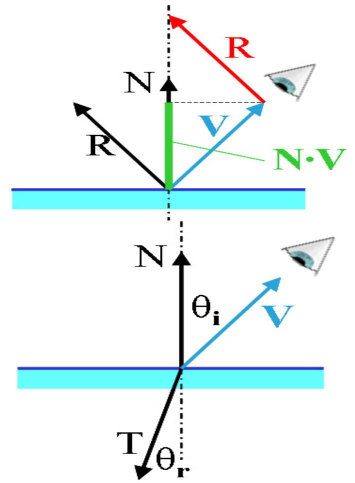
```
FOR all pixels P0 DO
 1. calculate viewing ray from the eye through P0
     intersect ray with all objects and choose closest intersection P
 2. FOR all light sources L DO
     intersect shadow ray P -> L with all objects
     IF no intersection between P and L THEN shading += influence of L
 3.IF P is on a reflective surface
   THEN trace secondary ray; shading += influence of reflection
 4.IF P is on a transparent surface
   THEN trace secondary ray; shading += influence of transparency
```

Usually the viewing coordinate system is set up so that the xy-plane is the image plane and the main viewing direction is along the negative z-axis. Rays are used in parameter form: starting point plus parameter times direction vector.



So primary rays have the form: **eye point + s·(pixel – eye point)**,
Shadow rays have the form: **surface point P + s·(light source position L – P)**,

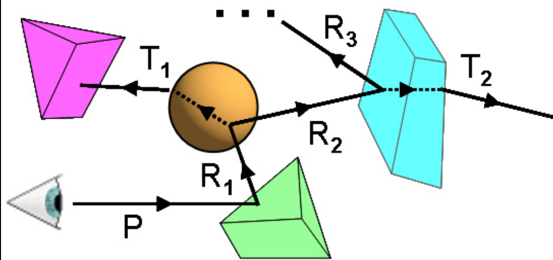
Reflection rays have the form: $\mathbf{P + s \cdot R}$, where $R = (2N \cdot V)N - V$ is the reflection direction of viewing ray V, according to the law of reflection (incident angle equals emergent angle). This calculation also guarantees that R has unit length (see figure).

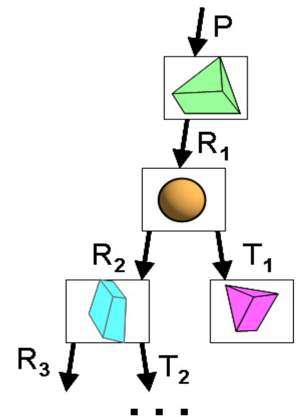


Transparency rays have the form $\mathbf{P + s \cdot T}$, where T results from applying Snell's law $\sin\theta_i : \sin\theta_r = \eta_r : \eta_i$ (η_r is the refraction index of material r):

$$T = -\frac{\eta_i}{\eta_r} V - (\cos\theta_r - \frac{\eta_i}{\eta_r}\cos\theta_i)N$$

Vector T has unit length as well (see figure).





When light rays are traced in reverse direction, as just explained, this generates a recursive function call sequence (see left figure), which corresponds to a "ray tree" (see right figure). Nonetheless, usually this tree is not stored explicitly; it is just a symbolic representation of the recursive call sequence.

Ray-Object Intersections

Objects which shall be rendered with ray-tracing have to fulfill only few prerequisites:

- it has to be possible to calculate an intersection point with a straight line,
- the surface normal at the intersection point has to be known,
- material properties (at this position) have to be available.

For BReps this is naturally simple, but also for CSG-trees, which can be evaluated recursively. Fortunately also many other data formats fulfill these prerequisites (e.g. freeform surfaces). For each kind of primitive a function has to be provided which calculates the intersection with a ray. As an example, we are going to describe this for a sphere and for a polygon in more detail:

Ray-Sphere Intersection

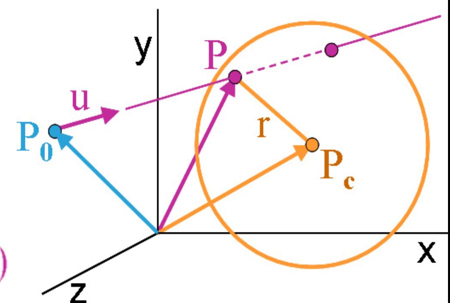
Sphere equation: $$|P - P_c|^2 - r^2 = 0$$

We substitute the ray's equation for **P**: $$|(P_0 + su) - P_c|^2 - r^2 = 0$$

For better legibility we introduce **ΔP**: $$\Delta P = P_c - P_0$$

We obtain a quadratic equation: $$s^2 - 2(u \cdot \Delta P)s + (|\Delta P|^2 - r^2) = 0 \qquad (u^2 = 1)$$

The 2 solutions correspond to the 2 intersection points with the sphere: $$s = u \cdot \Delta P \pm \sqrt{(u \cdot \Delta P)^2 - |\Delta P|^2 + r^2}$$



In cases when $r^2 << |\Delta P|^2$ (which happens quite often), the solutions of the equations suffer from numerical instability. To prevent that from happening, we can take advantage of the fact that $u^2=1$ and rewrite the equation. The new equation is numerically more stable:

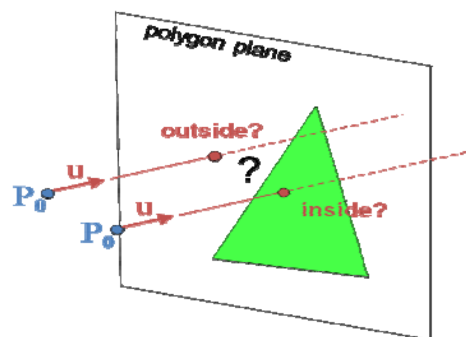
$$\mathbf{s = u \cdot \Delta P \pm \sqrt{r^2 - |\Delta P - (u \cdot \Delta P)u|^2}}$$

Ray-Polygon Intersection

When intersecting a ray with a polygon, we first test if the polygon is

front-facing in the first place (see Backface Detection). Then we substitute the ray equation $P = P_0 + s\cdot u$ into the polygon's plane equation $Ax + By + Cz + D = 0$ and since $N=(A,B,C)$, we can also write it as $N\cdot P = -D$: $N\cdot(P_0 + s\cdot u) = -D$. From that we get $s = -(D + N\cdot P_0)/N\cdot u$ which, when substituted into the ray equation, gives us the intersection point with the polygon plane.

Now we must check if the intersection point actually lies within the polygon edges or outside the polygon boundary (see figure).

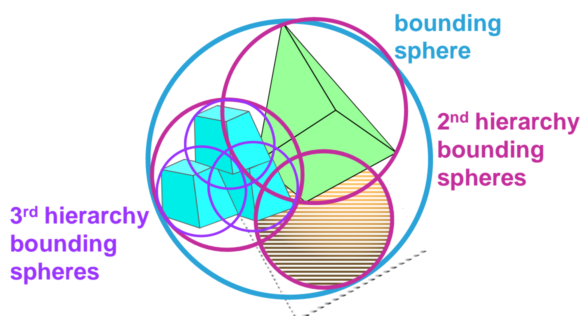


Ray-Tracing Acceleration

Ray-tracing is computationally very expensive. To render a scene with only 1000 polygons or objects to a 1000x1000 pixel image plane requires 10^9 intersection calculations for the primary rays only, without optimizations. Therefore it is necessary to accelerate the technique significantly. The most important method to achieve this is to reduce the number of necessary intersection calculations by exploiting coherence.

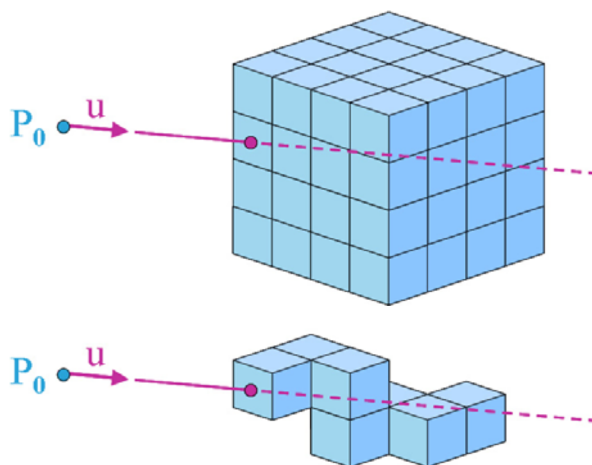
Bounding Volumes

Before intersecting all parts of a more complex object (or some part of the scene) with a ray, it can be checked if the ray comes anywhere near to the object altogether. To accomplish this, *bounding spheres*, which enclose the whole complex object, are added to the data structure. Rays that do not intersect that bounding sphere will for sure not to hit the object, so no intersection calculations have to be performed with any of the object's parts. This concept can be used hierarchically, i.e. complex subparts of objects may again have bounding spheres themselves and so on, until we reach simple objects. In this way the cost for the intersection tests can be reduced from O(n) to approximately O(log n). Instead of bounding spheres any other bounding volumes can be used, for example bounding boxes. Note that often there is a trade-off between better-fitting enclosing shapes and the effort for testing rays against this bounding volume.



Space Subdivision Methods

Alternatively, we can subdivide the whole space in which the scene resides with a regular grid. It does not make much difference whether the sub-cubes are stored in an array or in an octree. Now, only objects lying in the subspaces which the ray traverses need to be intersected with the ray. So we need a fast method to determine the next sub-cube along the ray path. 3D Bresenham algorithms lend themselves to this task. As soon as an intersection point in a sub-cube is found, the search for further intersection points can be stopped.



For individual sub-cubes we can also apply the following procedure:

The ray $P = P_0 + s\cdot u$ enters the sub-cube at P_{in}. The normal vectors of the cube's faces are (1,0,0), (–1,0,0), (0,1,0), (0,–1,0), (0,0,1), (0,0,–1). For the three faces with $u\cdot N > 0$ (the other three faces are irrelevant!) we determine the intersection point with the ray and choose the closest one (smallest s). This method works also for cubes with varying sizes (as in octrees).

$$P_{out,k} = P_{in} + s_k u$$

$$N_k \cdot P_{out,k} = -D_k$$

$$s_k = \frac{-D_k - N_k \cdot P_{in}}{N_k \cdot u}$$