Computer Graphics

Lecture 6: ray tracing

Ágoston Sipos
siposagoston@inf.elte.hu

Eötvös Loránd University
Faculty of Informatics

2025-2026. Spring semester

Table of contents

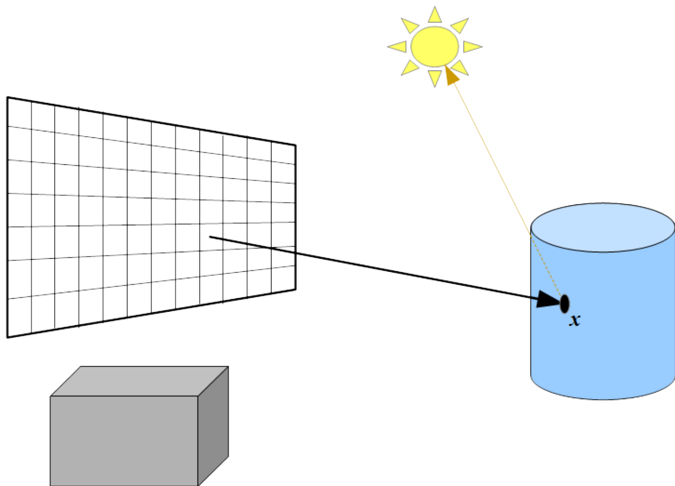
Recursive ray tracing
- Motivation
- Illumination equation
- Remarks

Ray tracing acceleration
- Motivation
- Bounding volumes

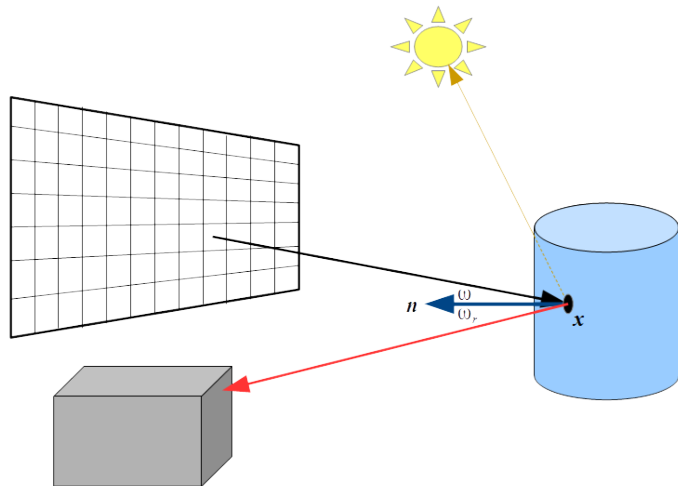
Space partitioning methods
- Uniform grid
- Octree
- k-d tree

Ray tracing in hardware

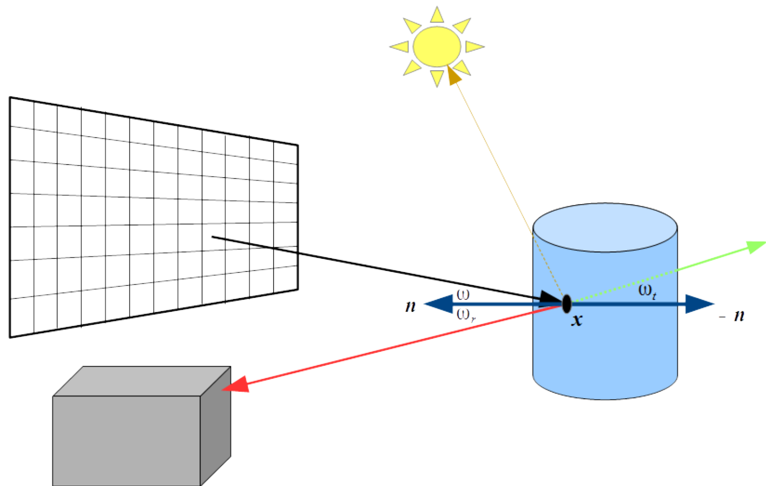
Recursive ray tracing – shadow ray



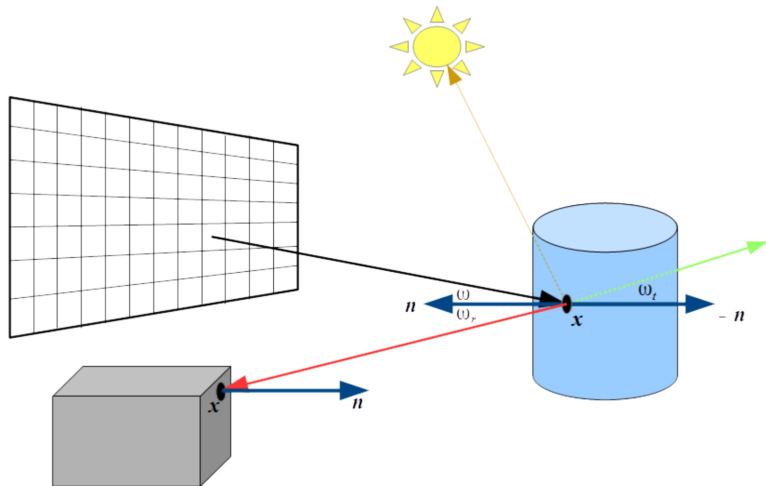
Recursive ray tracing – ideal reflection



Recursive ray tracing – ideal refraction

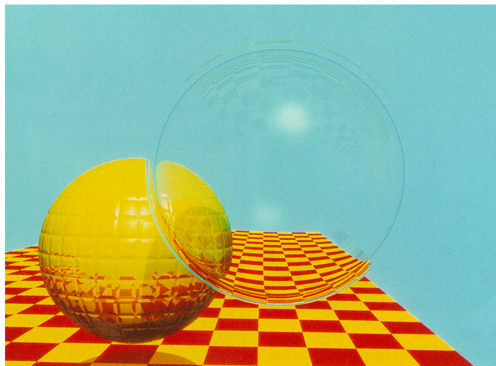


Recursive ray tracing – recursion



Recursive ray tracing

For each pixel, we determine their color independently – we solve the *shadow* and *visibility* tasks.

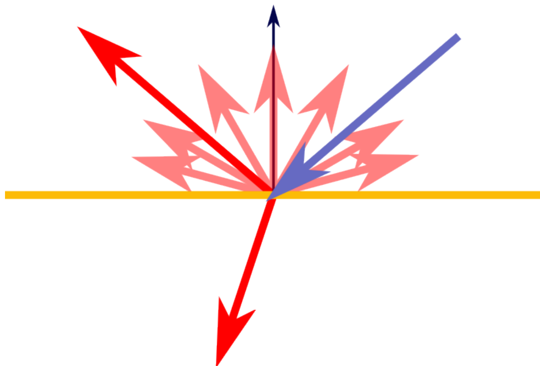


Turner Whitted, 1980

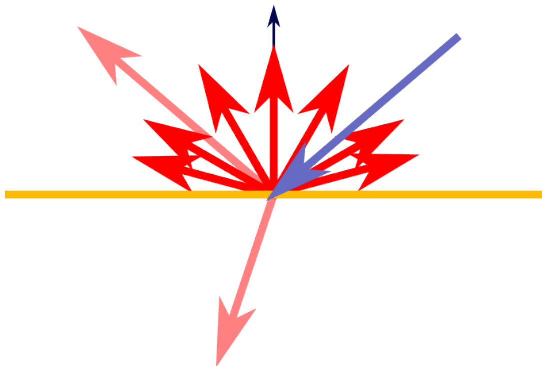
Light components

- We divide the path of the light into two components:
  - *Coherent* component:
    - Ideal reflection and refraction according to optics
    - We continue to follow the path of the light
  - *Incoherent* component:
    - Everything else
    - Of these, we only consider the direct illumination of the abstract light source

Coherent component



Incoherent component



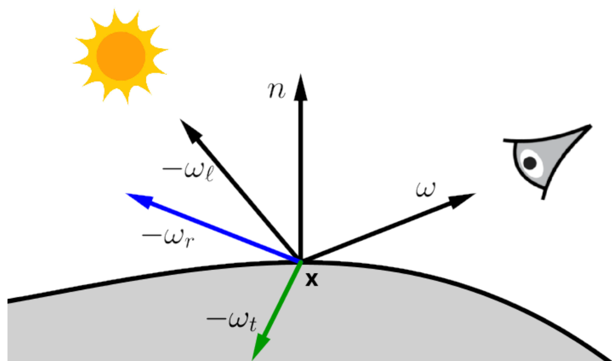
Notations

- For simplicity, the scalar product of two vectors will now be denoted by $\mathbf{a} \cdot \mathbf{b}$
- For directions we use letters ω, ω' etc. but they are still unit vectors i.e $\omega \in \mathbb{R}^3 : |\omega| = 1$

Simplified illumination equation

We solve the following, simplified illumination equation:

$$L(\mathbf{x} \to \omega) =$$
$$L_e(\mathbf{x} \to \omega) + k_a \cdot L_a + \sum_{\ell \in \text{Lights}} f_r(\omega_\ell \to \mathbf{x} \to \omega) L_i(\omega_\ell \to \mathbf{x})(-\omega_\ell \cdot \mathbf{n})$$
$$+ k_r \cdot L(\mathbf{x} \leftarrow \omega_r) + k_t \cdot L(\mathbf{x} \leftarrow \omega_t)$$



Ray tracing

$$L(\mathbf{x},\omega) = L_e(\mathbf{x},\omega) + k_a \cdot L_a + \sum_{\ell \in \text{Lights}} f_r(\mathbf{x},\omega_\ell,\omega) L_i(\mathbf{x},\omega_\ell)(-\omega_\ell \cdot \mathbf{n}) + k_r \cdot L(\mathbf{x}_r,\omega_r) + k_t \cdot L(\mathbf{x}_t,\omega_t)$$

The radiance emitted from the surface point $\mathbf{x}$ in the ω direction

- From the eye position, we emit rays through each pixel (for example, through the center of the pixel)
- The direction of the rays is denoted by $-\omega$ (minus omega!)
- The closest intersection of the ray and the objects of the scene gives $\mathbf{x}$

Emission

$$L(\mathbf{x},\omega) = L_e(\mathbf{x},\omega) + k_a \cdot L_a + \sum_{\ell \in \text{Lights}} f_r(\mathbf{x},\omega_\ell,\omega) L_i(\mathbf{x},\omega_\ell)(-\omega_\ell \cdot \mathbf{n}) + k_r \cdot L(\mathbf{x}_r,\omega_r) + k_t \cdot L(\mathbf{x}_t,\omega_t)$$

This term describes the surface's own radiation – *emission* – from the surface point $\mathbf{x}$ in the ω direction

Ambient light

$$L(\mathbf{x},\omega) = L_e(\mathbf{x},\omega) + k_a \cdot L_a + \sum_{\ell \in \text{Lights}} f_r(\mathbf{x},\omega_\ell,\omega) L_i(\mathbf{x},\omega_\ell)(-\omega_\ell \cdot \mathbf{n}) + k_r \cdot L(\mathbf{x}_r,\omega_r) + k_t \cdot L(\mathbf{x}_t,\omega_t)$$

$k_a \in [0,1]$ is the surface's, $L_a \in \mathbb{R}_0^+$ is the environment's *ambient* coefficient.
The *ambient* part of the equation approximates the amount of light that is generally present, reaching all surfaces, regardless of their position and not dependent on abstract light sources. Its purpose is to replace the amount of light lost due to approximations.

Light sources

$$L(\mathbf{x},\omega) = L_e(\mathbf{x},\omega) + k_a \cdot L_a + \sum_{\ell \in \text{Lights}} f_r(\mathbf{x},\omega_\ell,\omega) L_i(\mathbf{x},\omega_\ell)(-\omega_\ell \cdot \mathbf{n}) + k_r \cdot L(\mathbf{x}_r,\omega_r) + k_t \cdot L(\mathbf{x}_t,\omega_t)$$

- The summation term encompasses the considered incoherent reflections
- We only take into account the direct effect of light sources
- And only if the light source is visible from the **x** surface point

Light sources

$$L(\mathbf{x},\omega) = L_e(\mathbf{x},\omega) + k_a \cdot L_a + \sum_{\ell \in \text{Lights}} f_r(\mathbf{x},\omega_\ell,\omega) L_i(\mathbf{x},\omega_\ell)(-\omega_\ell \cdot \mathbf{n}) + k_r \cdot L(\mathbf{x}_r,\omega_r) + k_t \cdot L(\mathbf{x}_t,\omega_t)$$

- ω_ℓ is the unit vector from the light source to the surface point.
- $f_r(\mathbf{x},\omega_\ell,\omega)$ is now just the BRDF with diffuse and specular reflection. (later...)
- $-\omega_\ell \cdot \mathbf{n}$ is the cosine of the angle between the surface normal and the vector pointing towards the light source $\approx$ in unit time, how many photons from ω_ℓ hit the surface

Light sources

$$L(\mathbf{x},\omega) = L_e(\mathbf{x},\omega) + k_a \cdot L_a + \sum_{\ell \in \text{Lights}} f_r(\mathbf{x},\omega_\ell,\omega) L_i(\mathbf{x},\omega_\ell)(-\omega_\ell \cdot \mathbf{n}) + k_r \cdot L(\mathbf{x}_r,\omega_r) + k_t \cdot L(\mathbf{x}_t,\omega_t)$$

- If the luminous intensity of the ℓ light source towards us is Φ_ℓ and its position is $\mathbf{x}_\ell$ then

$$L_i(\mathbf{x},\omega_\ell) = v(\mathbf{x},\mathbf{x}_\ell) \cdot \frac{\Phi_\ell}{\|\mathbf{x}-\mathbf{x}_\ell\|^2}.$$

- $v(\mathbf{x},\mathbf{x}_\ell) \in [0,1]$ visibility function: *Is there anything between* $\mathbf{x}$ *and light source?*

Light sources – squared decay

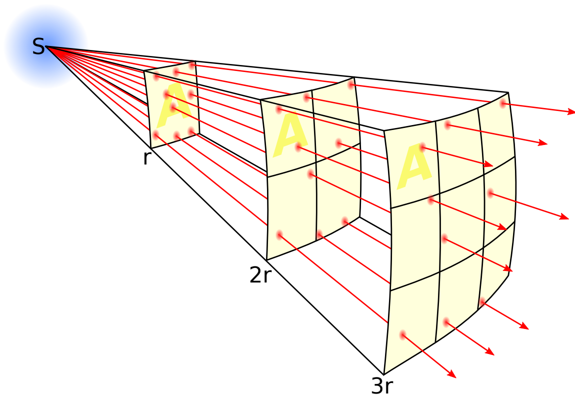
- Why do we divide by the squared distance between the intersection points and the light source?
- Let us consider a point light source that emits L radiance uniformly in all directions
- Moving away from it, how much light would we measure per square centimeter on the sphere inscribed around the point light source? radiance divided by the surface area of the sphere corresponding to the distance, i.e

$$L_r = \frac{L}{4\pi r^2}$$

- So the ratio of the amount of light measured at two different distances:

$$\frac{L_{r_1}}{L_{r_2}} = \frac{\frac{L}{4\pi r_1^2}}{\frac{L}{4\pi r_2^2}} = \frac{r_2^2}{r_1^2}$$

Light sources – squared decay



Wikipedia

Light sources

$$L(\mathbf{x},\omega) = L_e(\mathbf{x},\omega) + k_a \cdot L_a + \sum_{\ell \in \text{Lights}} f_r(\mathbf{x},\omega_\ell,\omega) L_i(\mathbf{x},\omega_\ell)(-\omega_\ell \cdot \mathbf{n}) + k_r \cdot L(\mathbf{x}_r,\omega_r) + k_t \cdot L(\mathbf{x}_t,\omega_t)$$

$v(\mathbf{x},\mathbf{x}_\ell) \in [0,1]$ function

- $= 0$, if the light source is not visible from $\mathbf{x}$,
- $= 1$, if visible,
- $\in (0,1)$, if there are transparent objects in between.
- To calculate v, we launch a so-called *shadow ray* from $\mathbf{x}$ towards $\mathbf{x}_\ell$ and look at its intersection with the objects.

Reflection

$$L(\mathbf{x},\omega) = L_e(\mathbf{x},\omega) + k_a \cdot L_a + \sum_{\ell \in \text{Lights}} f_r(\mathbf{x},\omega_\ell,\omega) L_i(\mathbf{x},\omega_\ell)(-\omega_\ell \cdot \mathbf{n}) + k_r \cdot L(\mathbf{x}_r,\omega_r) + k_t \cdot L(\mathbf{x}_t,\omega_t)$$

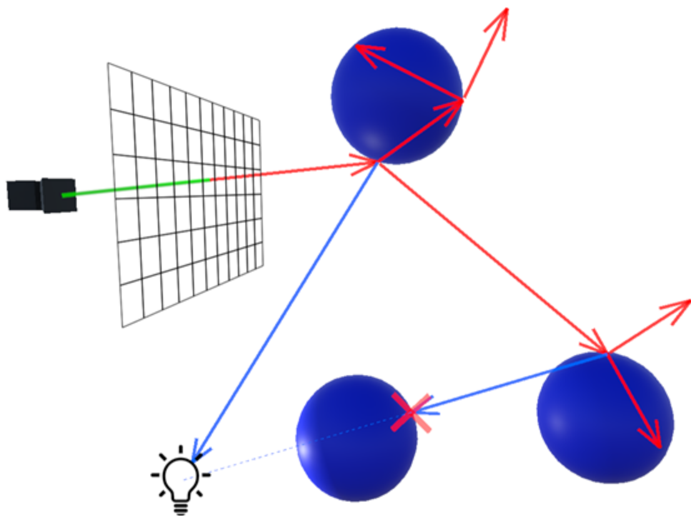
- We consider the light arriving from the mirror direction in proportion to k_r (Fresnel coefficient)
- ω_r is the incident vector corresponding to the ideal mirror direction.
- $\mathbf{x}_r$ is the nearest intersection point between the objects of the scene and the ray with $\mathbf{x}$ origin and $-\omega_r$ direction
- Calculating $L(\mathbf{x}_r,\omega_r)$ is the same as calculating $L(\mathbf{x},\omega)$ (recursion!).
- New ray: starts from $\mathbf{x}$ instead of eye position, with $-\omega_r$ direction.

Refraction

$$L(\mathbf{x},\omega) = L_e(\mathbf{x},\omega) + k_a \cdot L_a + \sum_{\ell \in \text{Lights}} f_r(\mathbf{x},\omega_\ell,\omega) L_i(\mathbf{x},\omega_\ell)(-\omega_\ell \cdot \mathbf{n}) + k_r \cdot L(\mathbf{x}_r,\omega_r) + k_t \cdot L(\mathbf{x}_t,\omega_t)$$

- We consider the light arriving from the refraction direction in proportion to k_t (Fresnel coefficient)
- ω_t is the incident vector corresponding to the ideal refraction direction.
- $\mathbf{x}_t$ is the nearest intersection point between the objects of the scene and the ray with $\mathbf{x}$ origin and $-\omega_t$ direction
- Calculating $L(\mathbf{x}_t,\omega_t)$ is again the same as calculating $L(\mathbf{x},\omega)$ (recursion!).
- New ray: starts from $\mathbf{x}$ instead of eye position, with $-\omega_t$ direction.

Recursive ray tracing – coherent component



Remarks – recursion

- After finding the first intersection, the goal is the compute the light from that point towards the camera – this is described by the illumination equation.
- On the right hand side of the equation, the coherent component uses the same L function (with different arguments) – this is where the recursion happens via starting new rays.

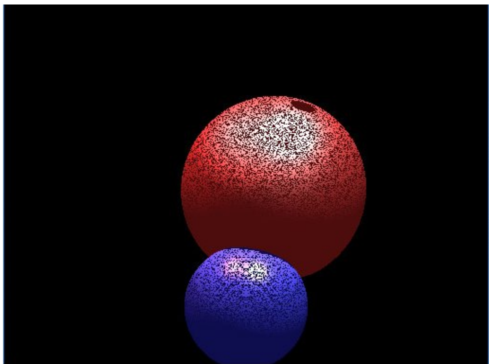
$$L(\mathbf{x}, \omega) = (\dots) + \sum_{\ell \in \text{Lights}} (\dots) + k_r \cdot L(\mathbf{x}_r, \omega_r) + k_t \cdot L(\mathbf{x}_t, \omega_t)$$

- Stopping the recursion is usually done with a maximum depth.
- Recursion may also be stopped earlier if the contribution of the next ray to the final amount of light is negligible – i.e. when the product of the relevant k_r, k_t coefficients is small.
- Computing the contribution v of light sources also works via starting rays (shadowrays), however, this is not recursive!

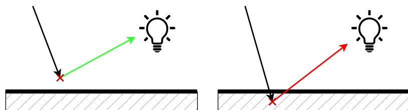
Remarks – colors

- The formulas only calculate the amount of light, so we actually calculated the light reaching the camera at a single wavelength
- Almost everything depends on the wavelength:
  - Light emitted by the surface and light sources: L_e, L_i
  - The reflective property of the surface (BRDF approximate and coherent coefficients): f_r, k_a, k_r, k_t
  - The ideal refraction direction: ω_t
- So we need to calculate it for all wavelengths in the visible light spectrum
- Instead, we usually do it on just a few wavelengths
- In practice, it is often only on the RGB color triad

Remark – self-shadowing



Remark – self-shadowing

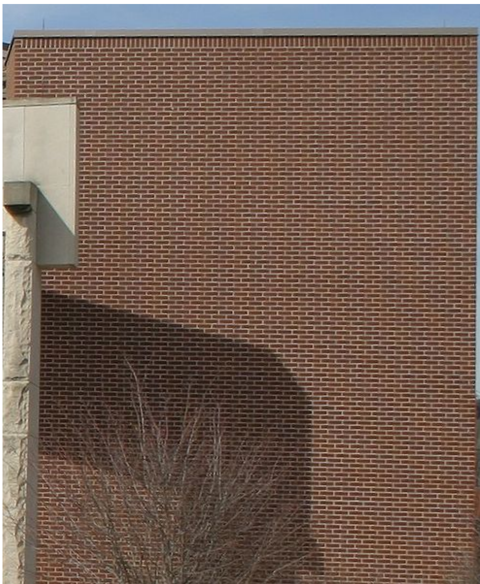


- Due to numerical inaccuracies, the point of intersection may actually be slightly inside the body
- In this case, the rays shot towards the light sources may hit the starting surface!
- Solution: shift the starting point of the ray in its normal direction
- Let's also pay attention to what exactly is needed: the position(s) of the intersection or is it enough that we know there is an intersection?

Remark – aliasing

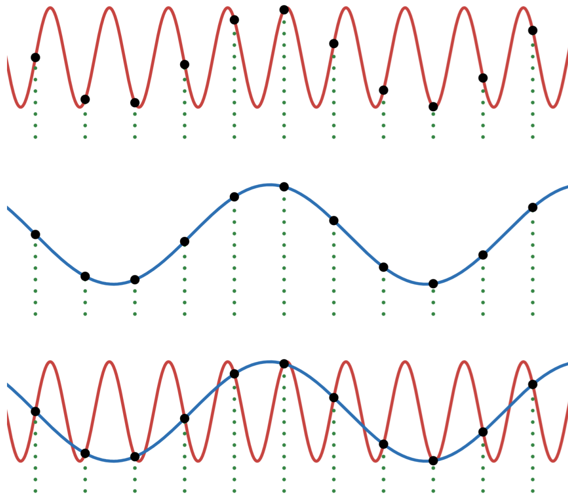
- Essentially, we are sampling point by point $\rightarrow$ any changes faster than the sampling frequency are *aliased*, i.e. we register them as non-existent, lower-frequency signal components

Remark – aliasing

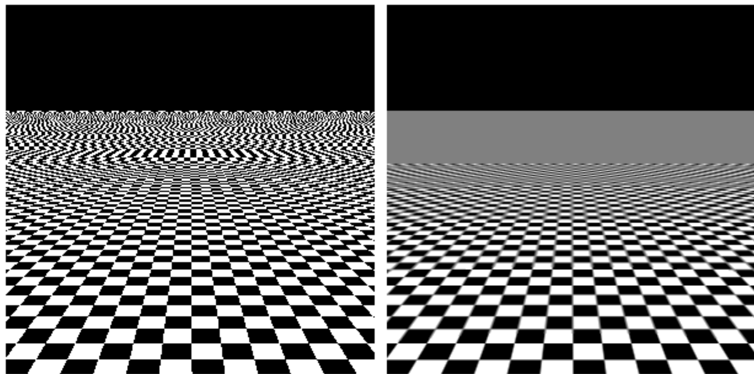


Remark – aliasing

Remark – aliasing



Sampling in graphics – alias



https://textureingraphics.wordpress.com/what-is-texture-mapping/anti-aliasing-problem-and-mipmapping/

- Left: alias
- Right: a possible solution using Mipmaps (in a later lecture)

Sampling in graphics – alias

- So far, we have only launched a single ray per pixel („per window”).
- If we launch more, the Nyquist frequency increases
- Only one color is needed per pixel $\rightarrow$ the different colors brought in by the rays have to be summed up somehow (e.g. averaged)
- But the results of several rays can be summed up in several ways (that is: we can filter)

Sampling in graphics – alias

- The alias cannot be eliminated by uniform sampling (only if the incoming signal is guaranteed not to contain too high frequency components)
- However, if the sampling is not uniform, but is done according to a proper distribution, then instead of the alias, there will be random, high-frequency noise in the image
- Our eyes have already adapted to this!

Remark

- Notice that: above we treated light as a particle
- For this reason, we cannot reproduce phenomena arising from the wave nature of the light (interference, diffraction, etc.).
- When do these matter?
  - Because of interference, we see the peacock's feathers or the surface of the soap bubble in the color we see it in
  - And diffraction plays a role in some of the subtle shadow phenomena

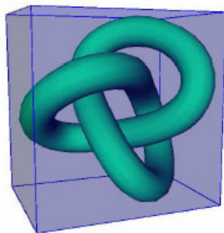
Acceleration of intersection test – motivation

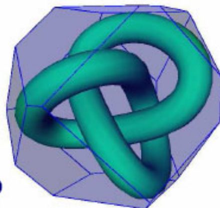
- The speed of the algorithm mostly depends on the speed of the *intersection test*
- How can we speed this up?
- Do not intersect with objects that definitely won't be hit by the ray!
- Do not intersect with objects that certainly will give an intersection point, that is further than the one already found!

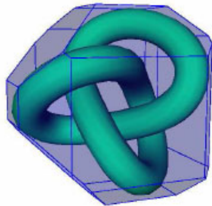
Bounding volumes

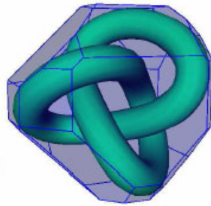
- Let's surround each object with some kind of *volume*, which can be used to quickly calculate intersections.
- If a ray intersects the object, it must also intersect the volume $\rightarrow$ therefore if the volume is not intersected, then it is not necessary to check intersection with the object
- It will be the most efficient if the reverse is also likely to happen. (The closer the bounding volume is to the object)
- Bounding sphere: solving a quadratic equation.
- Bounding box: its edges are parallel to the axes, can be calculated quickly, see previous lecture!

Additional bounding volumes – convex polyhedrons









DOP: discrete oriented polytope

Additional bounding volumes – convex polyhedrons

- An n-sided convex polyhedron can be expressed as the intersection of n half-spaces (half-space: $ax + by + cz + d \leq 0$)
- That is, the planes of the side faces can be represented with normals pointing outwards

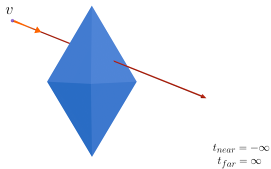
$$a_i x + b_i y + c_i z + d_i = 0\ ,\ \ i = 1.., n$$

- The ray-AAB intersection seen earlier can easily be generalized for this case!
- Let ray $\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{v}$

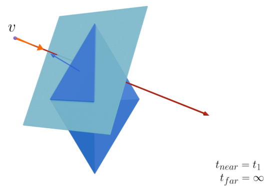
Ray intersection with convex polyhedron

1. Let $t_{near} := -\infty$, $t_{far} := \infty$
2. For every $i \in \{1, 2, .., n\}$ face: let the face's equation be $a_i x + b_i y + c_i z + d_i = 0$, such that the $\mathbf{n}_i = [a_i, b_i, c_i]^T$ normal of the face points outward from the polyhedron
   2.1 Calculate the intersection point t_i of the ray and the plane of the face
   2.2 **If** $\mathbf{v} \cdot \mathbf{n}_i < 0$, **then** $t_{near} := \max\{t_{near}, t_i\}$
   2.3 **Else** $t_{far} := \min\{t_{far}, t_i\}$
3. **If** $t_{near} > t_{far}$, **then** there is no intersection
4. **Else** the ray steps into at t_{near} steps out at t_{far} from the convex polyhedron (i.e. the ray intersects the convex polyhedron if $[t_{near}, t_{far}] \cap \mathbb{R}^+ \neq \emptyset$)

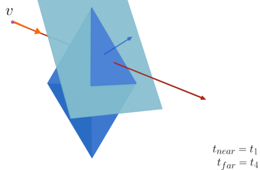
Ray intersection with convex polyhedron



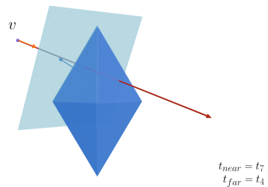
$i = 1$



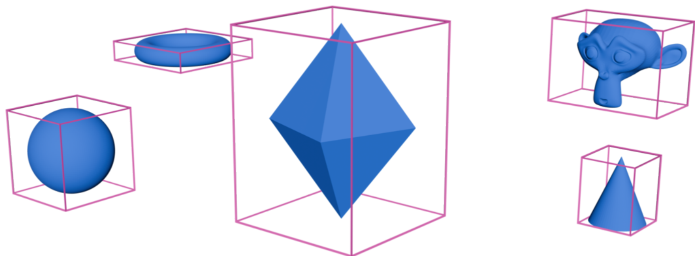
$i = 4$



$i = 7$



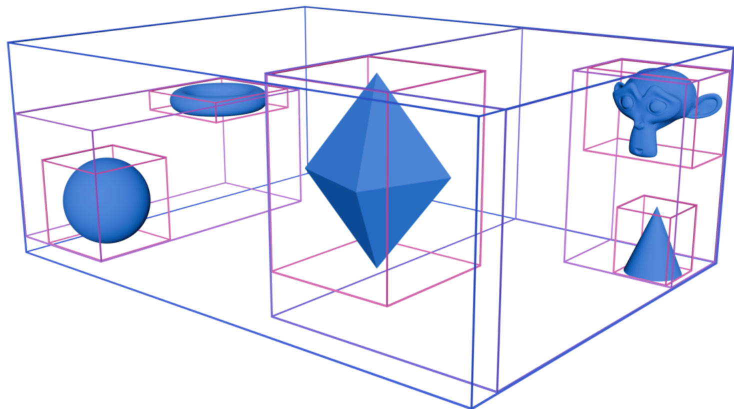
Using bounding volumes



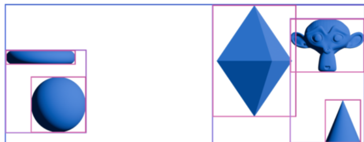
Bounding volume hierarchy

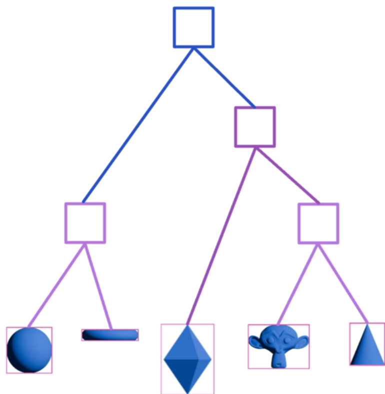
- We bound multiple volumes with larger ones.
- We get a tree structure.
- A subtree should only be evaluated if there is an intersection with its root.

Bounding volume hierarchy



Bounding volume hierarchy

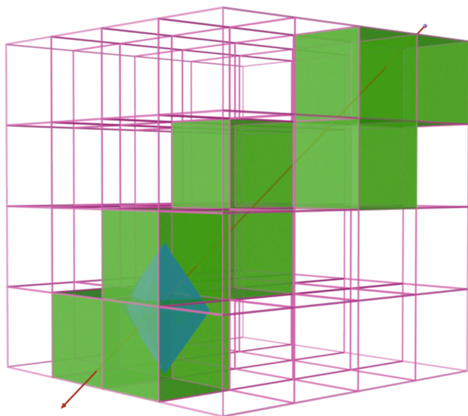




Uniform grid partitioning

- We cover the entire scene with a uniform 3D grid.
- Pre-processing: for each cell, we record the included objects.
- Usage: ray intersection is only performed with those objects whose cells the ray passes through
- Advantage: the cells to be examined can be calculated quickly with a line drawing algorithm! (See later on rasterization)
- Disadvantage: unnecessarily many cells – most of them cover empty space.

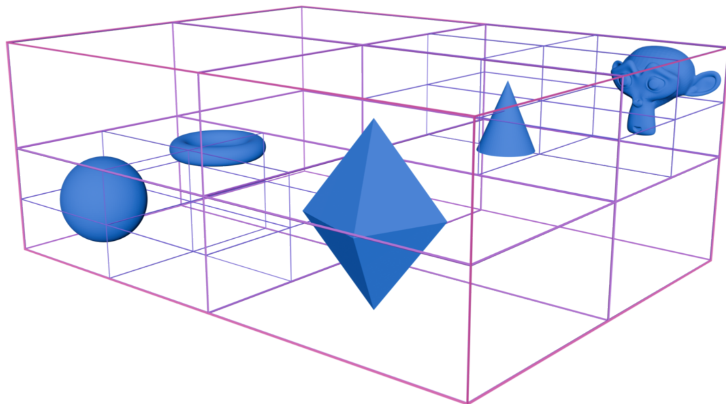
Uniform grid partitioning



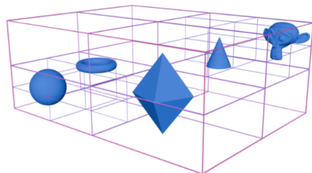
Octree

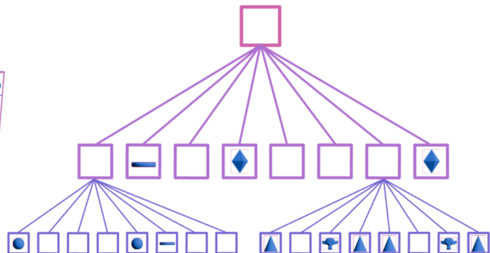
- Root: bounding box with edges parallel to the axes (*AABB*) encompassing the entire scene
- Let's cut this into eight equal parts!
- For each new box: if there are *enough* objects in it, we divide further, otherwise we stop.
- Advantage: we do not divide the empty parts further unnecessarily.
- Disadvantage: more complicated traversal.
- Disadvantage: a branch can become too deep $\rightarrow$ in practice, a predefined depth is also given, when it's reached we do not divide the cells any further (even when it contains more objects than the maximum)

Octree



Octree

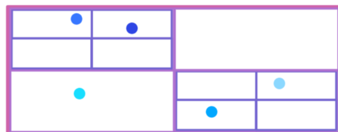


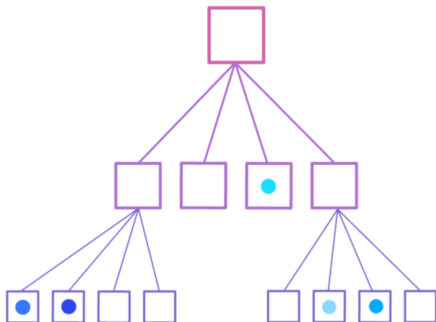


Quadtree

- Octree in a plane
- The current cell is always divided into four equal parts, parallel to the axes

Quadtree

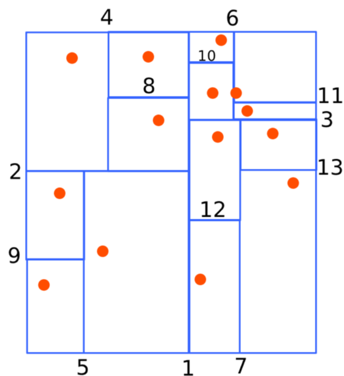


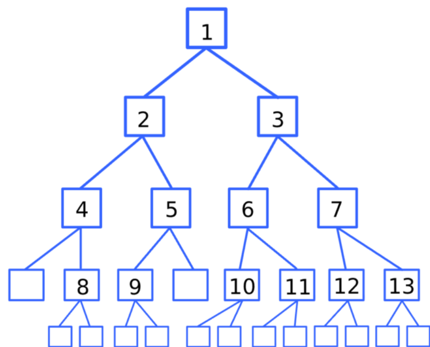


k-d tree

- Problem with octree: it always cuts in the middle and along all planes – it doesn't take objects into account.
- Octree: search time $\approx$ height of the tree. BUT! the octree is unbalanced.
- k-d tree: in each step, we cut with a single plane that is perpendicular to an axis.
- Order: $X, Y, Z, X, Y, Z, \ldots$.
- Placement of bisecting plane based on heuristics:
  - spatial median method
  - body median method
  - cost model-based method

k-d tree

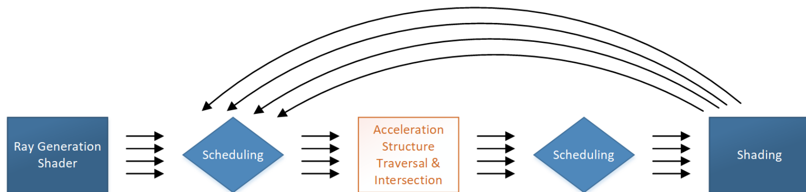




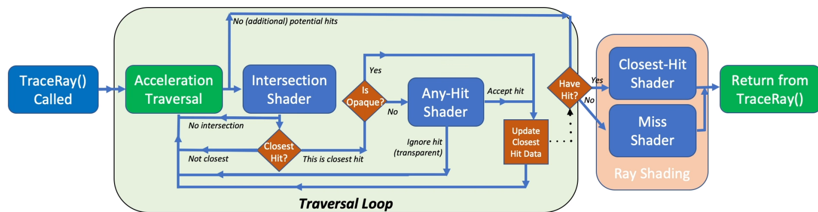
*DXR

- Since 2018, hardware-accelerated ray tracing has been available in NVIDIA GPUs
- This became part of the DX12 standard
- There is support with Compute shader fallback, down to the Pascal architecture
- But real hardware ray tracing is only on Volta, Turing and Ampere and newer architectures

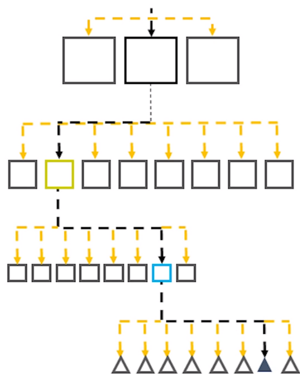
*HW RT

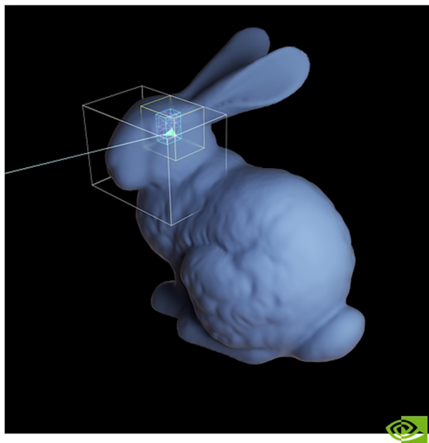


*HW RT

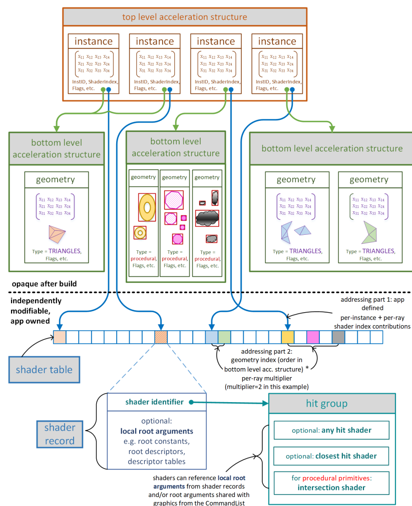


*HW RT – Bounding Volume Hierarchy





*HW RT



*DXR 1.0

5 new types of shaders:

- **Raygeneration shader**: shader generating the rays
- **Intersection shader**: for procedural geometries; there is a built-in ray-triangle intersection, you don't have to write your own (and probably you don't want to - see Watertight Ray/Triangle Intersection)
- **Closest hit shader**: essentially the fragment shader equivalent; a shader that runs once on the nearest intersection
- **Anyhit shader**: callable shader to decide transparency during traversal; can be invoked multiple times; the order of invocations is not guaranteed
- **Miss shader**: if there were no intersected geometry, then this is called (here you can calculate, for example, sky color or whatever you want)

*DXR 1.1

What is really interesting is that from here you can call the TraceRay command in any shader (not only from the graphics assembly line, but also from the compute shader!).
Also, there is DispatchIndirect, which gives you a lot of new options.