Lecture 12: Ray-tracing based rendering

DHBW, Computer Graphics

Lovro Bosnar

8.3.2023.

Syllabus

- 3D scene
  - Object
  - Camera
  - Light
- Rendering
  - Ray-tracing based rendering
  - Rasterization-based rendering
- Image and display

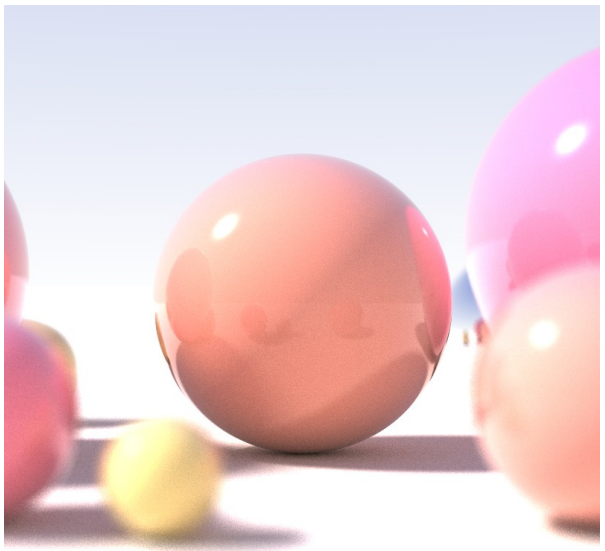
- Ray-tracing based rendering
  - Overview
  - Camera rays
  - Intersections (visibility)
  - Shading
  - Light transport

Ray-tracing-based rendering:
https://www.realtimerendering.com/Real-Time_Rendering_4th-Real-Time_Ray_Tracing.pdf

Ray-tracing based rendering overview

Introduction

- Ray-tracing is a method **inspired by physics of light** → realistic image synthesis
- Ray-tracing is considered one of the **most elegant techniques** in computer graphics
- Many phenomena such as **shadows, reflections and refracted light are intuitive and straightforward** to implement



Robert L. Cook Thomas Porter Loren Carpenter. "Distributed Ray-Tracing" (1984).

https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

Rendering recap

- **Rendering: calculate a color of each pixel of the virtual image plane**
- **Rendering process: visibility and shading**
  - **Visibility**: find objects that are visible from camera
    - Find which surfaces are visible, ignore participating media
  - **Shading**: what is the color of visible objects
    - Use incoming light, view, object shape and material information

Ray-tracing for visibility

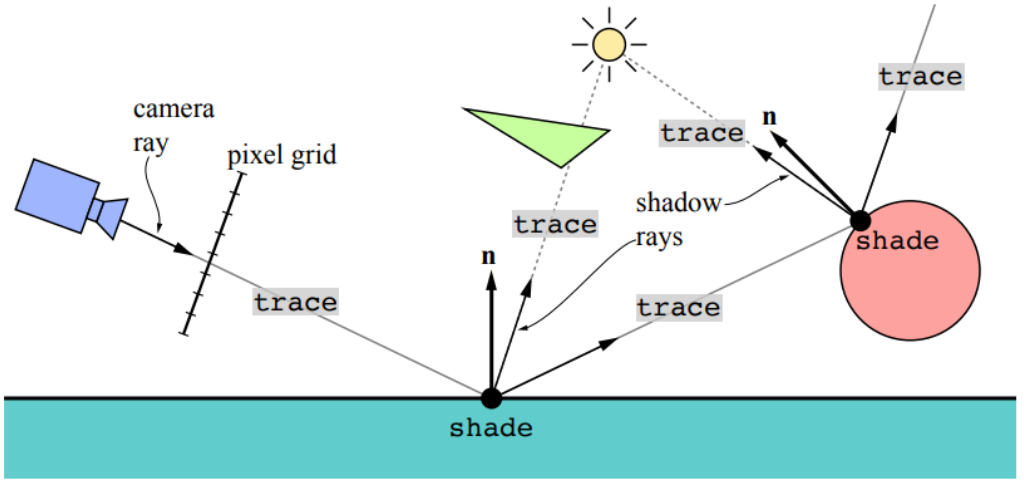
- **Visibility problem**: two points are visible to each other if line segment that joins them does not intersect any obstacle.
- Ray-tracing fundamentally **solves visibility problem using ray-casting**
  - **Finding objects visible from camera**: generate ray from camera and test intersections with objects
  - **Shading** requires finding from where light might be incident to surface by generating rays from shading point into 3D scene - **light transport**

Ray-tracing-based rendering

- **Generate ray** for each pixel of virtual image plane → **camera ray**
  - **Perspective camera**: generate rays from aperture to each pixel
  - **Orthographic camera**: generate parallel rays for each pixel
- **Ray-object intersection**: testing intersection of generated ray and 3D scene objects to find closest intersection → objects visible from camera
- **Shading**: calculating the color and intensity of intersected points
  - Light-matter calculation: light absorption and reflection
  - Light transport for computing incoming light: tracing rays to potential sources of light

Ray-tracing-based rendering: trace & shade

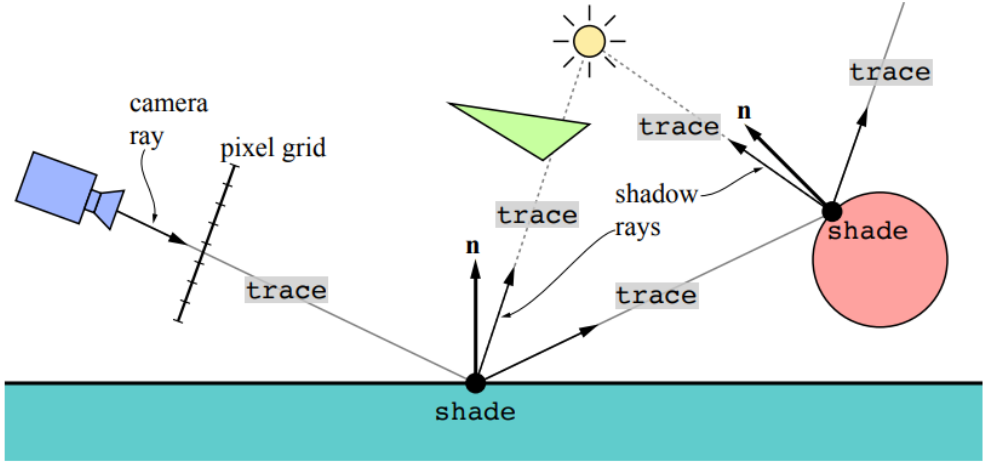
- Raytracing can be described with two functions **`trace()`** and **`shade()`**
- **`trace()`** is geometrical part of algorithm responsible for finding closest intersection between ray and the objects in 3D scene
- **`Shade()`** returns color of the ray intersecting object found by **`trace()`**



Ray-tracing-based rendering:
https://www.realtimerendering.com/Real-Time_Rendering_4th-Real-Time_Ray_Tracing.pdf

`trace()` for camera rays

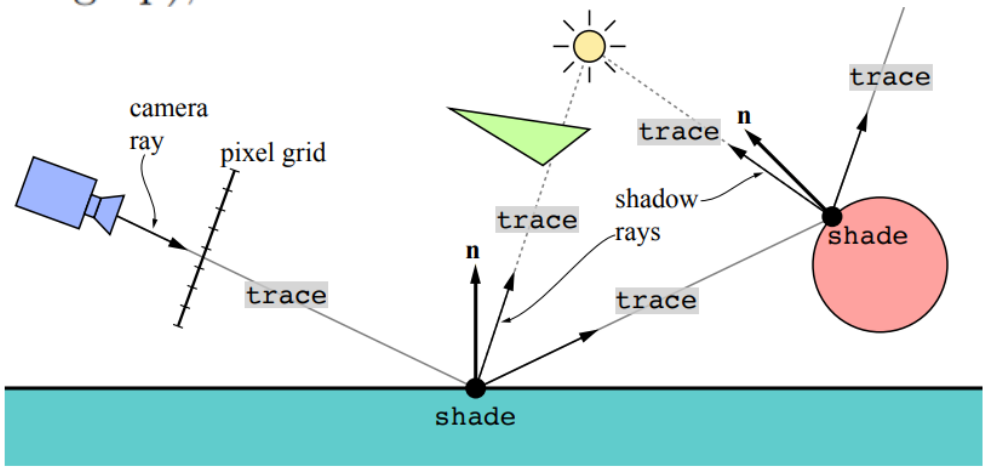
- **Rendering starts by generating camera rays for each pixel in the image**
- **`trace()` function is used on generated camera rays**
  - Find closest intersection of camera ray with 3D scene objects
  - Naive `trace()` function: loop through all n objects in the scene and returns closest intersection
    - `O(n)` performance. Spatial acceleration structure (BVH or k-d tree) → `O(log(n))` performance



`trace()` for camera rays

```
function RAYTRACEIMAGE
    for p do in pixels
        color of p = TRACE(camera ray through p);
    end for
end function
```

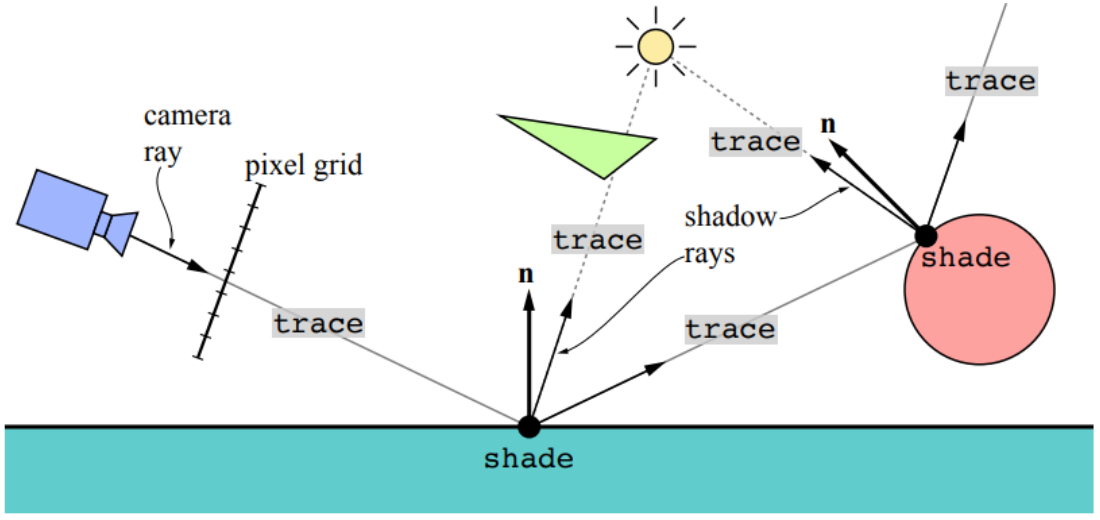
```
function TRACE(ray)
    pt = find closest intersection;
    return SHADE(pt);
end function
```



Ray-casting is often used in `trace()` function for determining visibility between two points.

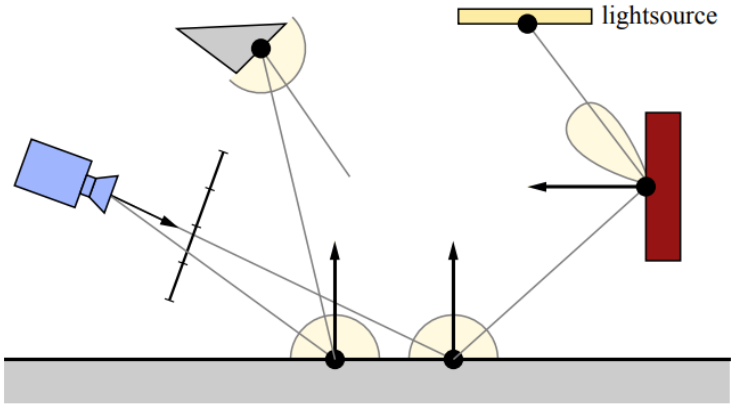
`shade()` for intersections

- `shade()` calculates color in intersection.
- `shade()` can be arbitrarily complex:
  - It can just return the color of object
  - It can use material information in intersected point with incoming light information
    - Use `trace()` for closest light sources
    - Use `trace()` to gather incoming light from all directions (e.g., other surfaces)



`trace()` for light transport

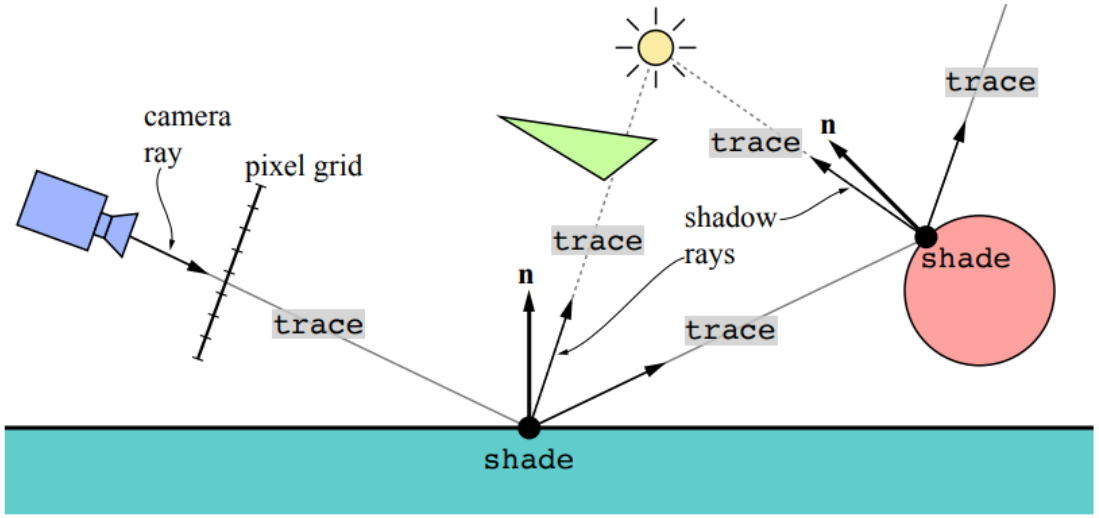
- **Light transport: obtaining incident light on shading point using `trace()`**
  - Calculate **visibility between shaded point and light source (direct illumination)**
  - Compute light from **perfect specular or diffuse surfaces (Whitted ray-tracing)**
  - Compute **indirect light reflected from other surfaces**



`shade()` for intersections

```
function SHADE(point)
    color = 0;
    for L do in light sources
        TRACE(shadow ray to L);
        color += evaluate material;
    end for
    color += TRACE(reflection ray);
    color += TRACE(refraction ray);
    return color;
end function
```

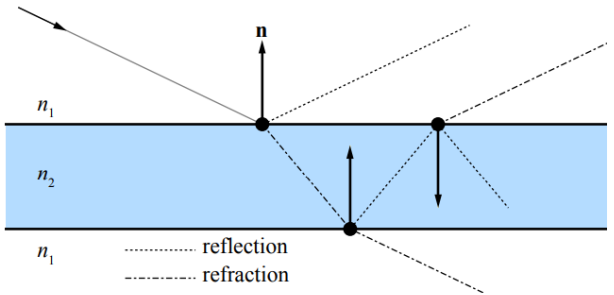
```
function TRACE(ray)
    pt = find closest intersection;
    return SHADE(pt);
end function
```



- Each `shade()` can call `trace()` and each `trace()` can call `shade()`
- **Ray depth** is term with indicates number of rays that have been shot recursively along a ray path

`shade()` and `trace()`

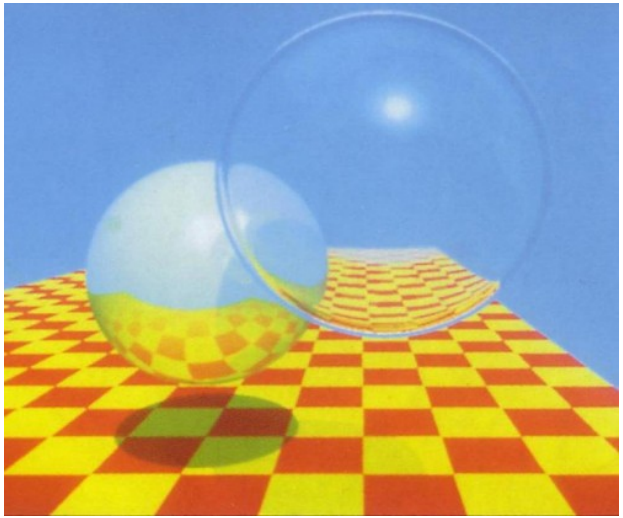
- `shade()` function is implemented by user as a **shader** program
  - Material (scattering functions and textures) are used to define light-surface interaction – amount of absorption and reflection, that is, color
- Traversal and intersection testing which takes place in `trace()` function is often implemented on CPU
  - GPU rendering uses compute or ray-tracing shaders in Vulkan or DXR



`shade()` defines interaction of light with surface, its color.

Ray-tracing structure

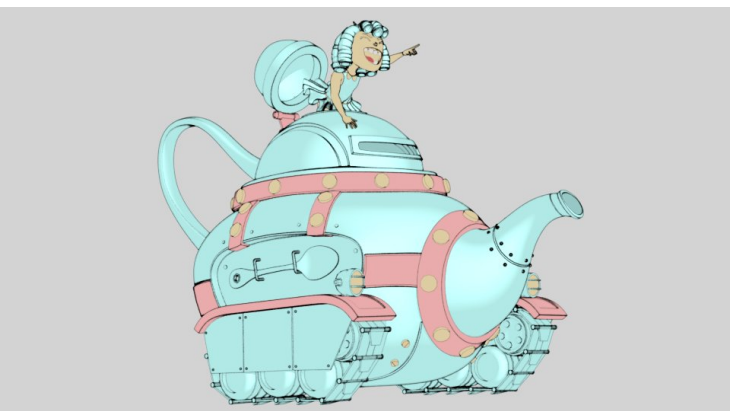
- Ray-tracing structure consisting of `shade()` and `trace()` is basis for **Whitted ray-tracer.**
  - Assumptions: surface are perfectly sharp (specular) reflections and refractions or diffuse.
- Whitted ray-tracing is foundation of many other rendering variants such as path-tracing which are solving rendering equation and global illumination.



(Non-)Photo-realistic rendering

- **Level of photo-realism depends on reproducing the color and intensity of objects - appearance**
- Shading plays crucial role for object appearance:
  - **Photo-realistic rendering**
  - **Expressive or artistic rendering ➞ non-photo-realistic rendering**
- Photo-realistic rendering is useful for understanding physically-based shading from which non-photo realistic rendering can be derived with means of exaggeration

https://www.artstation.com/artwork/rANRe5



"Rolling Teapot" - Model by Brice Laville, concept by Tom Robinson, render by Esteban Tovagliari - RenderMan 'Rolling Teapot' Art Challenge: https://appleseedhq.net/gallery.html#https%3A%2F%2Fappleseedhq.net%2Fimg%2Frenders%2Frolling-teapot.jpg

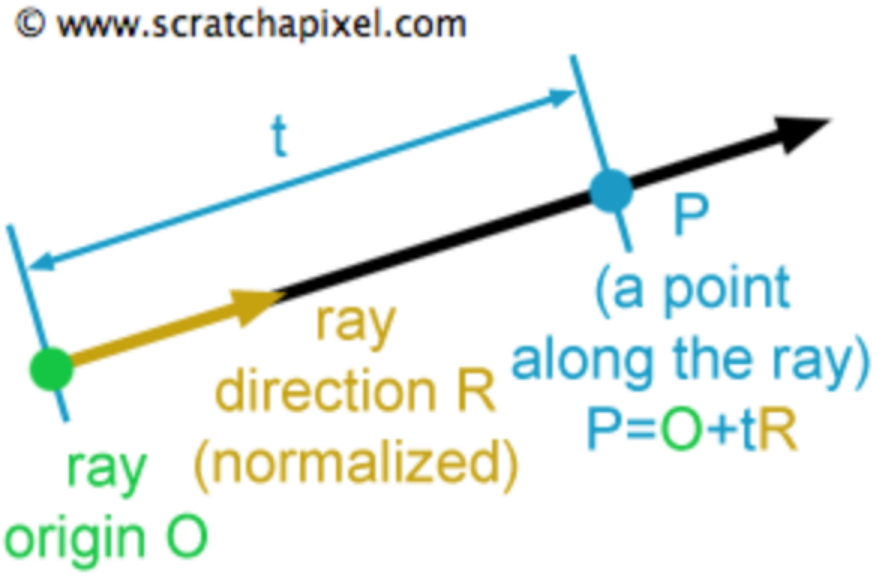
Ray-tracing based rendering

Ray

- **Ray is fundamental element of ray-tracing used for solving visibility**
- Ray is defined as:
  - `vector3f origin;`
  - `vector3f direction;`
- Any point on ray is is defined with parametric equation:

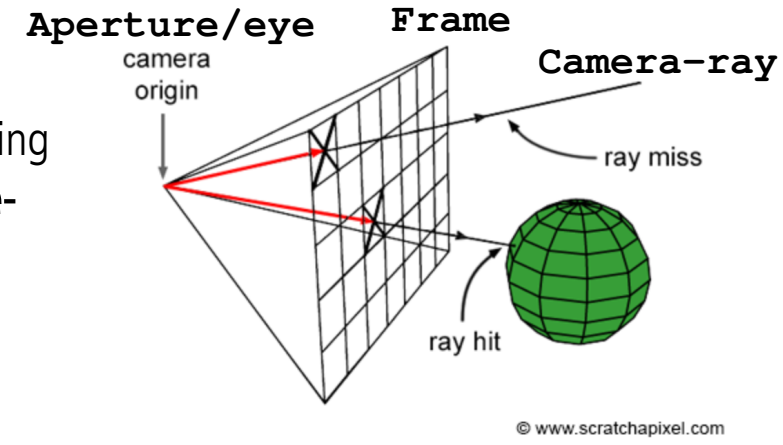
```
P(t) = origin + t * direction;
```

  - `t` – distance from `origin` to `P(t)`
  - **`t > 0`** → `P(t)` `is` in front of ray's `origin`
  - `t < 0` → `P(t)` `is` behind the ray's `origin`



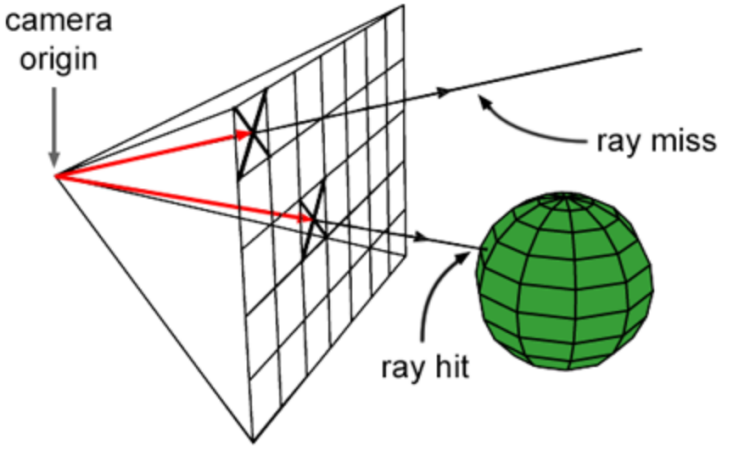
Generating camera rays

- Camera:
  - **Aperture** (eye) position and orientation of camera line of sight
  - **Field of view**: how much of the scene we see
  - **Frame**: array of pixels where image is formed
- **Camera rays** are generated starting from camera aperture and passing through each pixel in the film plane into 3D scene → **backward/eye-tracing**
  - Used to compute the visible objects → **ray-casting**
- Camera rays are also known as: **primary ray, view ray**



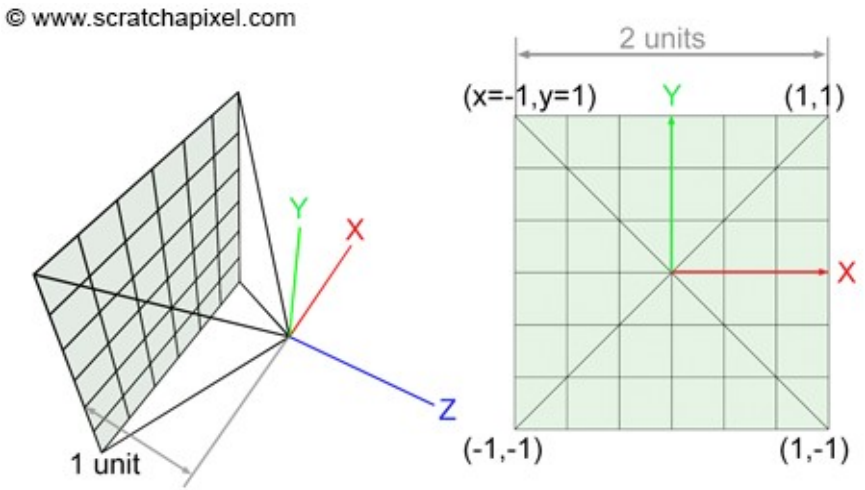
Generating camera rays

```
Vector3f ImageBuffer[imageWidth, imageHeight];
For (int j = 0; j < imageHeight; ++j)
{
    For (int i = 0; i < imageWidth; ++i)
    {
        Ray ray = buildCameraRay(i, j);
        For  (int k = 0; k < nObjectsInScene; ++k)
        {
            if (intersect(ray, object[k], intersectionContext)
            {
                // Object hit. Compute shading using intersectionContext.
                ImageBuffer[j * imageWidth + i] = shadingResult;
            }
            else
            {
                // Background hit. Compute background color...
                ImageBuffer[j * imageWidth + i] = backgroundColor;
            }
        }
    }
}
```



Generating camera rays: basic setup

- Assume:
  - Camera origin (aperture, eye) is in `(0,0,0)`
  - Camera is looking in negative `Z` axis
  - Film (image) plane is placed 1 unit from from camera's origin
  - Film dimensions are `2x2` units
  - Film is centered around `Z` axis
  - Image is square (`image_width == image_height`)
- **World-space ray is created by connecting world-space points**:
  - **Camera origin** – aligned with world coordinate origin
  - **Pixel center – requires transformation from raster space to world space**
- **World-space ray is needed for intersection since all objects in scene are also defined in world space**



https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays.html

Pixel center coordinates

- Pixel coordinates are originally defined in **raster space `[image_height, image_width]`**
  - Integer coordinates $(Pixel_x, Pixel_y)$ where left-top corner of frame is `(0,0)`
- Pixel position must be first normalized using frame dimensions giving **normalized device coordinates (NDC) [0,1]**

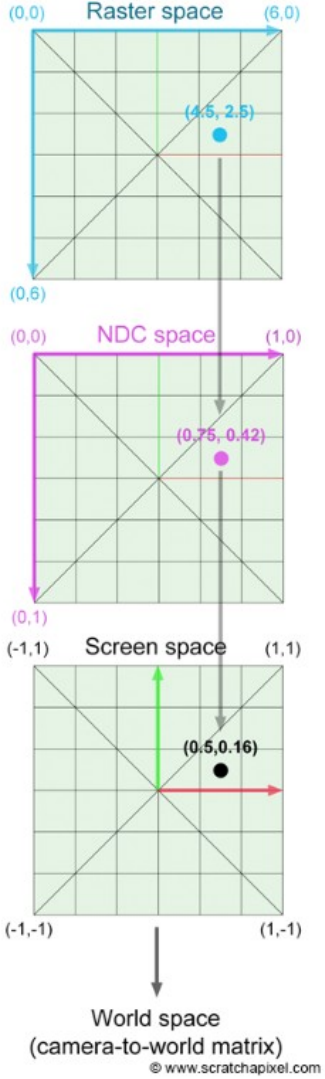
$$PixelNDC_x = \frac{(Pixel_x + 0.5)}{ImageWidth},$$

$$PixelNDC_y = \frac{(Pixel_y + 0.5)}{ImageHeight}.$$

- Finally, pixels are transformed from NDC to **screen (camera) space**

$$PixelScreen_x = 2 * PixelNDC_x - 1,$$

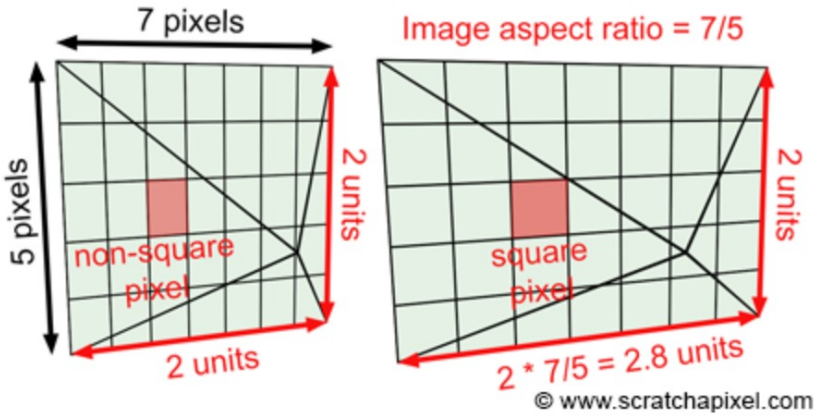
$$PixelScreen_y = 1 - 2 * PixelNDC_y.$$



Pixel center coordinates (frame with arbitrary aspect ratio)

- To ensure square pixels for image arbitrary aspect ratio, use image aspect ratio to scale frame size

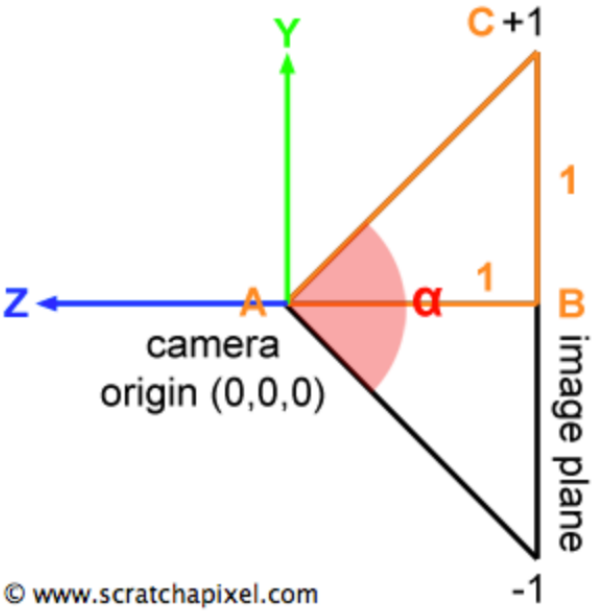
$$ImageAspectRatio = \frac{ImageWidth}{ImageHeight},$$
$$PixelCamera_x = (\quad PixelScreen_x \quad) * ImageAspectRatio,$$
$$PixelCamera_y = (\quad PixelScreen_y \quad).$$



Pixel center coordinates (arbitrary field of view)

- Field of view defines how much we see or zoom level.
- To incorporate arbitrary field of view:

$$PixelCamera_x = (\quad PixelScreen_x \quad) * ImageAspectRatio * tan(\frac{\alpha}{2}),$$
$$PixelCamera_y = (\quad PixelScreen_y \quad) * tan(\frac{\alpha}{2}).$$



$$|||BC|| = tan(\frac{\alpha}{2}).$$

Pixel center coordinates: camera space

- **Now, pixel coordinates are expressed in camera coordinate space**
  - Pixel coordinates are defined with regards to camera's image plane
- Currently, **camera is in default position** (camera coordinate system is aligned with world coordinate system)
  - Camera origin O (aperture): `(0,0,0)`
  - Orientation: negative Z axis
  - Image plane: 1 unit away from camera's origin
- Pixel coordinate position on the image plane

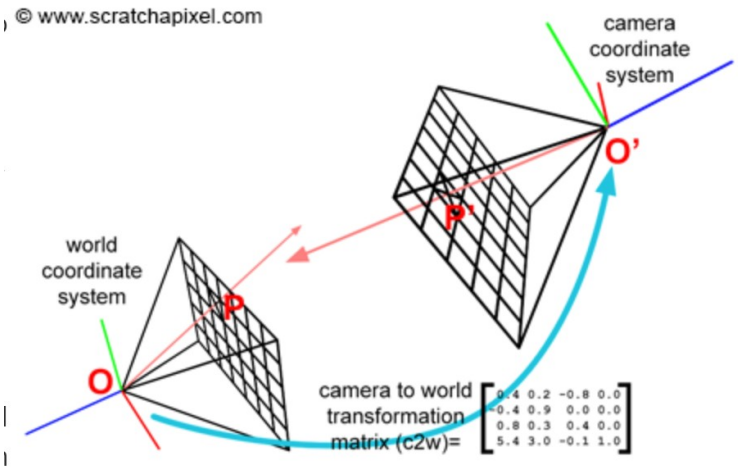
$$P_{cameraSpace} = (PixelCamera_x, PixelCamera_y, -1)$$

Generating camera rays: world space

- **Camera space ray** can be constructed using camera origin and pixel position in camera space

```
vector3 RayOrigin = cameraOrigin;
vector3 rayDirection = normalize(pixelCameraPosition – cameraOrigin);
```

- **World space ray** can be constructed:
  - `Camera-to-world matrix` is first applied applied on pixel position and camera origin
  - `vector3 RayOrigin = cameraOriginWorld;`
  - `vector3 rayDirection = normalize(pixelPositionWorld – cameraOriginWorld);`
- `Camera-to-world` can be constructed using `look-at` matrix.

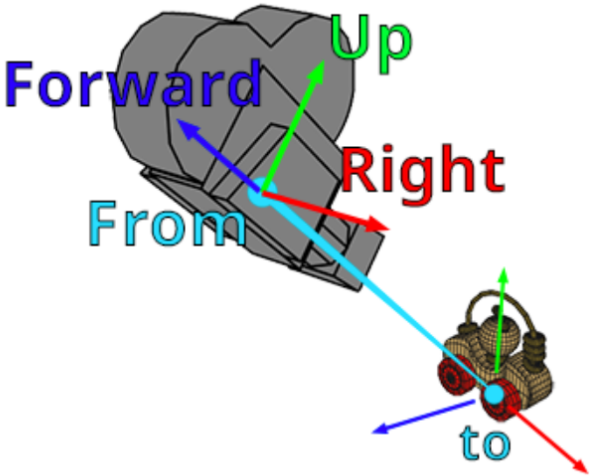


Camera is originally set in its default position. Camera-to-world matrix is used to move camera origin and pixel position for generating world space rays.

Look-at matrix: recap

- **4x4 transformation matrix** → transformation of camera from its local (camera) space to world space in 3D scene: **camera-to-world** and its inverse **world-to-camera** matrix

```
Forward = normalize(From - To)
Right = crossProduct(randomVec, Forward)
Up = crossProduct(forward, right)
```



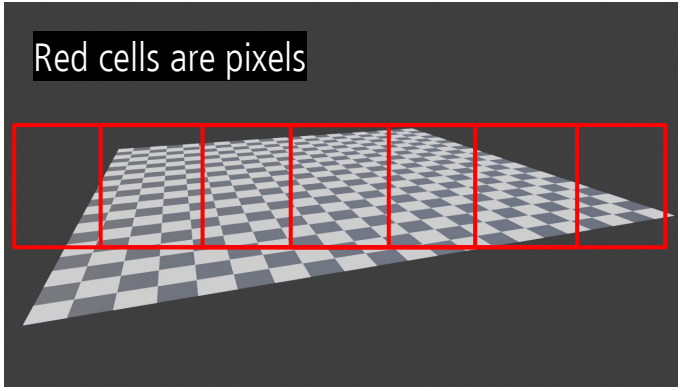
wwww.scratchapixel.com

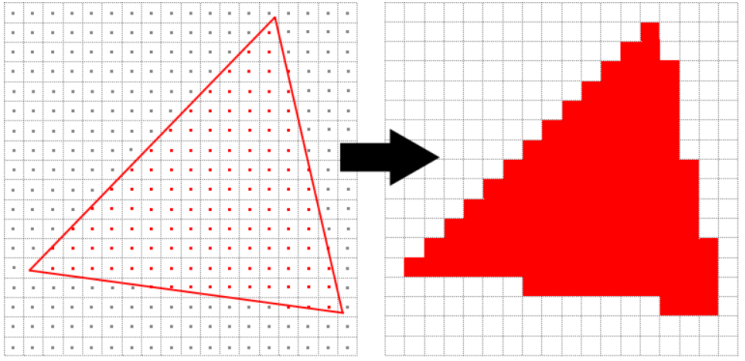
$$\begin{matrix} Right_x & Right_y & Right_z & 0 \\ Up_x & Up_y & Up_z & 0 \\ Forward_x & Forward_y & Forward_z & 0 \\ \texttt{From}_x & \texttt{From}_y & \texttt{From}_z & 1 \end{matrix}$$

`RandomVec = (0,1,0)` or other if `Forward` is close to `(0,1,0)` or `(0,-1,0)`

Generating multiple camera rays per pixel

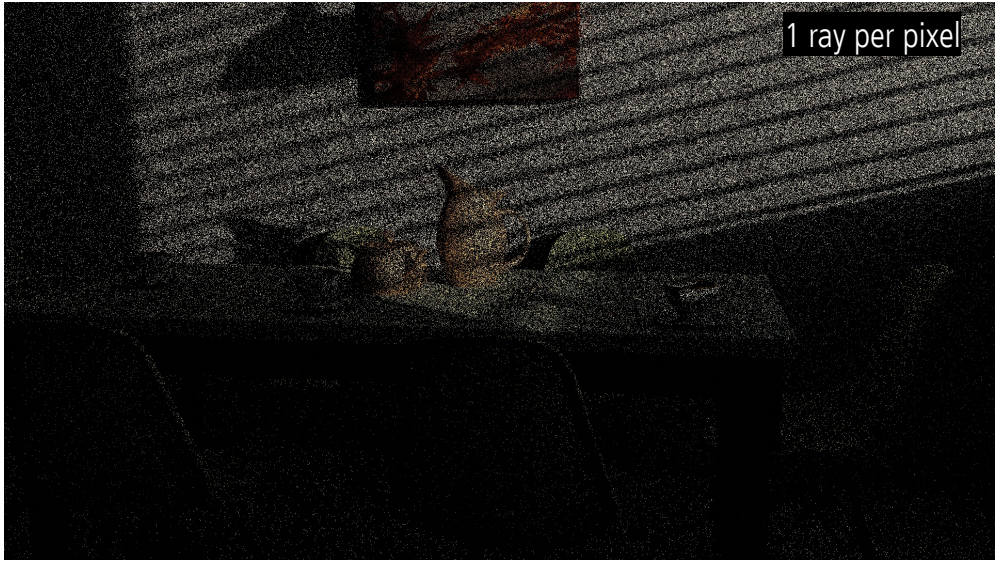
- Pixel footprint in 3D scene can cover large area with different textures and objects
  - Discretization leads to **aliasing**
- Since pixel can represent only one color, it is important to use multiple rays to obtain the color which is the most representative for the part of the scene covered by that pixel
  - Instead of pixel center, random points on pixel area are taken for building rays
  - Multiple rays per pixel are also called multiple samples per pixel (SPP)





Generating multiple camera rays per pixel

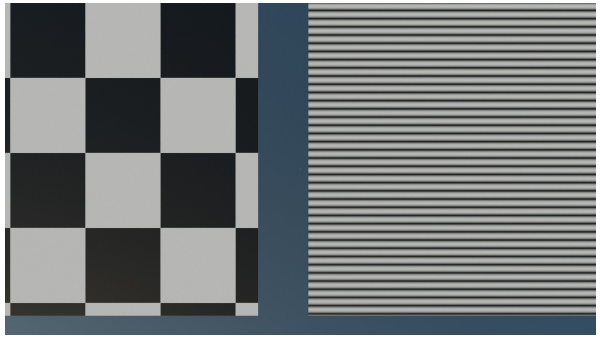
- Using multiple rays per pixels, **reduces noise**

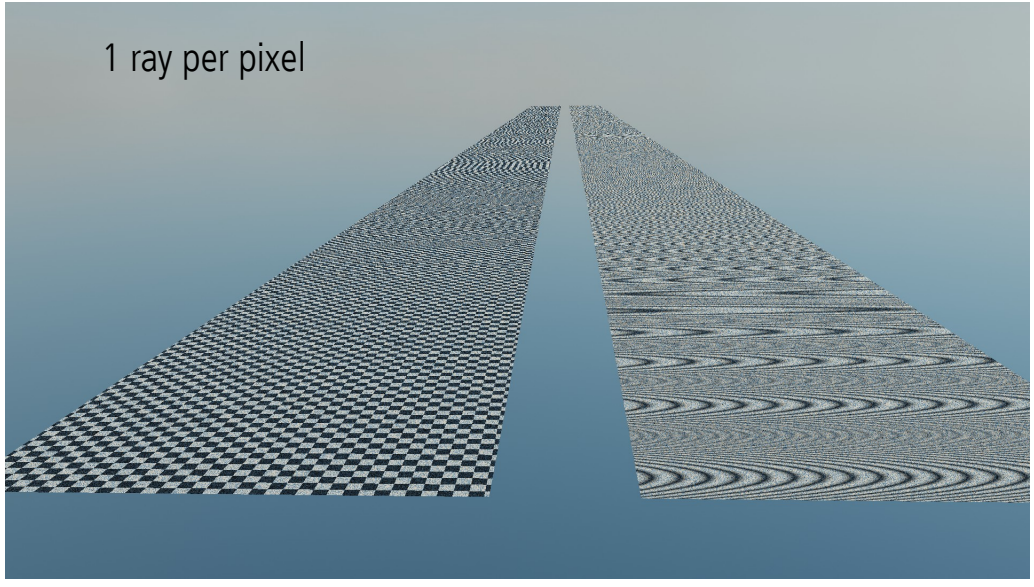


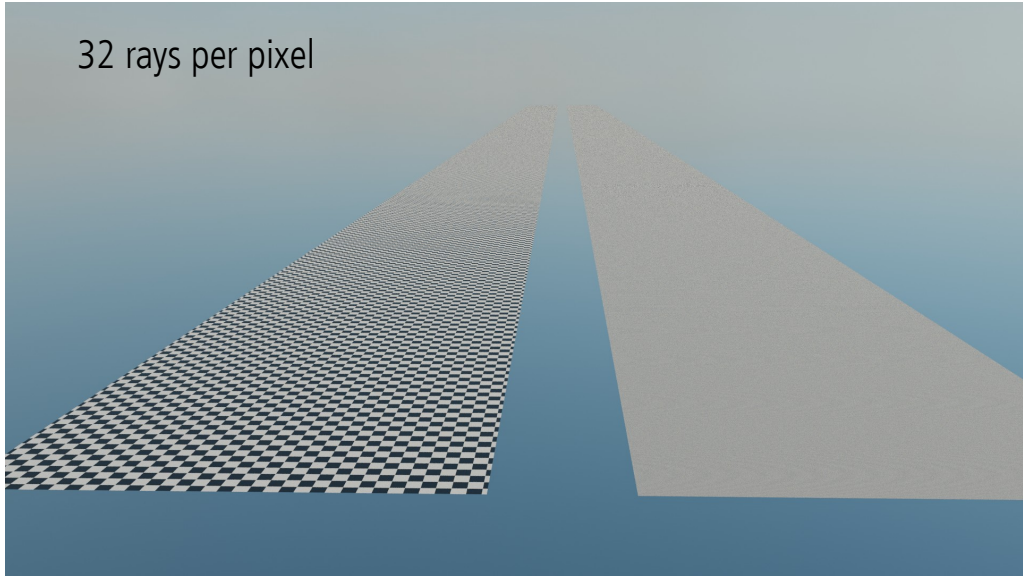
- Using multiple rays per pixel is especially important for methods building on raytracing and using stochastic sampling (e.g., Monte Carlo) such as path tracing.

Generating multiple camera rays per pixel

- Using multiple rays per pixels, **reduces aliasing**







Ray-tracing: image centric method

- Since rendering starts from virtual image plane, ray-tracing is called **image centric approach**
- **Once camera rays are generated, they are traced into scene and tested for intersections with objects**
  - Looping over all objects in 3D scene is performed and each is tested for intersection with current ray → **visibility test**
  - Testing intersections with objects **depends on object shape, i.e., triangulated mesh**

```
for P do in pixels
    for T do in triangles
        determine if ray through P hits T
    end for
end for
```

Camera ray-objects intersection testing

```
…
Ray ray = buildCameraRay(i, j);
for  (int k = 0; k < nObjectsInScene; ++k)
{
    if (intersect(ray, objects[k], intersectionContext)
    {
        // Object hit. Compute shading using intersectionContext.
        framebuffer[j * imageWidth + i] = shadingResult;
    }
    else
    {
        // Background hit. Compute background color...
        framebuffer[j * imageWidth + i] = backgroundColor;
    }
}
...
```

- `intersect()` method depends on object shape
- `ShadingResult` depends on object material.
- That is why decoupling material and shape of 3D object is useful.

Testing ray-object intersections

- Objects in 3D scene can be represented with different **shape (geometry) representations**
  - Parametric representations
    - Spheres, disks, planes, etc.
    - Surfaces and curves (e.g., Bezier curves, NURBS, etc.)
  - Implicit surfaces SDFs: spheres, cubes, etc.
  - Polygonal meshes (e.g., triangles, quads) and subdivision surfaces
  - Voxels
  - Etc.
- We will discuss:
  - **Ray-sphere intersection**
  - **Ray-triangle intersection and its extension to triangulated meshes**

Ray-sphere intersection

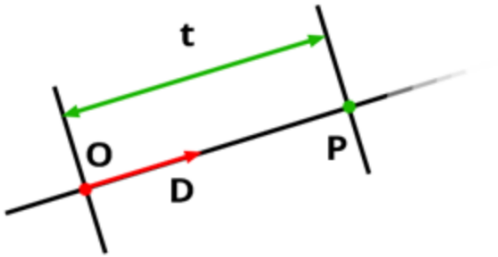
- Ray-sphere intersection is simplest ray-geometry intersection
- Parametric ray description: `P(t) = O + t * D`
- Implicit (algebraic) sphere form at world origin and radius R:

  $x^2 + y^2 + z^2 = R^2$

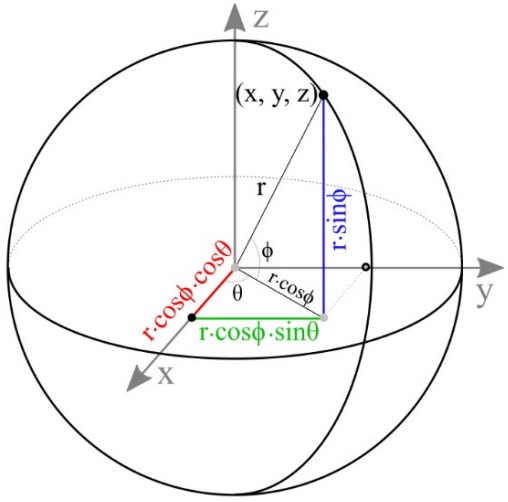
  - `x, y, z` are coordinates of a point on a sphere

  $P^2 - R^2 = 0$

  - Implicit function which defines implicit sphere shape
  - Equation is equal to zero if point `P` is on sphere



© www.scratchapixel.com



http://www.songho.ca/opengl/gl_sphere.html

Ray-sphere intersection

- Start with $P^2 - R^2 = 0$ and $P(t) = O + t * D$
- Substitute P with ray equation: $|O + t * D|^2 - R^2 = 0$
  - Develop: $O^2 + (Dt)^2 + 2ODt - R^2 = O^2 + D^2t^2 + 2ODt - R^2 = 0$
- Quadratic equation: $f(x) = ax^2 + bx + c$:
  - $a = D^2$, $b = 2OD$, $c = O^2 - R^2$. Solution:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
$$\Delta = b^2 - 4ac$$

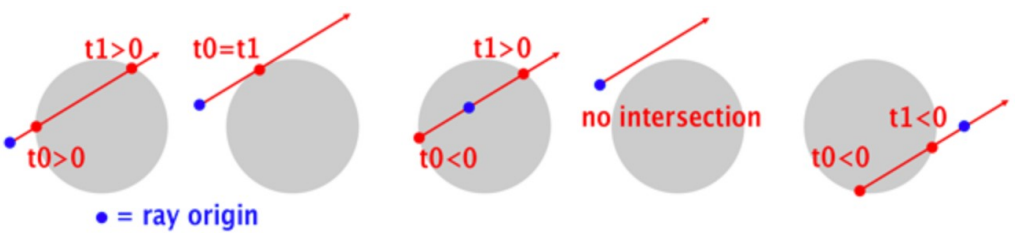
  - $\Delta > 0$: ray intersects sphere in two points (t_0 and t_1):

$$\frac{-b + \sqrt{\Delta}}{2a} \quad and \quad \frac{-b - \sqrt{\Delta}}{2a}$$

  - $\Delta = 0$: ray intersects sphere in one point ($t_0 = t_1$):

$$-\frac{b}{2a}$$

  - $\Delta < 0$: ray doesn't intersect the sphere



Ray-sphere intersection: arbitrary sphere position

- If sphere is translated from origin to point C, then:
  - $|P - C|^2 - R^2 = 0$
- Substituting the ray equation:
  - $|O + t * D - C|^2 - R^2 = 0$
- Solving quadratic equation gives t_0 and t_1:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$a = D^2 = 1$ (ray direction D is normalized)
$b = 2D(O - C)$
$c = |O - C|^2 - R^2$

Ray-sphere intersection: intersection point and context

- t_0, when inserted into ray equation `P(t) = O + t * D` gives closest **intersection point** of ray with a sphere

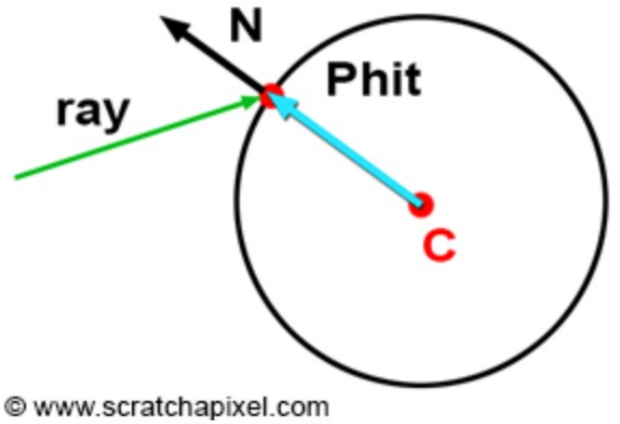
  `vector3 Phit = O + D * t0`

- Next to intersection point `Phit`, renderer often computes additional intersection information – **intersection context:**
  - **Normal** in intersection point
  - **Texture coordinate** in intersection point
  - Object/triangle index → material assigned to this object
  - Etc.
- Intersection context information calculation depends on object shape:
  - For parametric surfaces (u,v) parameters can be used as texture coordinates
  - For polygonal (e.g., triangle) mesh, texture coordinates must be precomputed and stored per vertex. Intersection point is used to interpolate texture coordinates per triangle face (e.g., barycentric interpolation)
- **Intersection context contains information which is used for shading of intersection point**

Ray-sphere intersection: normal

- Normal calculation in intersection point depends on shape representation
- For implicit sphere with center `C(x,y,z)`

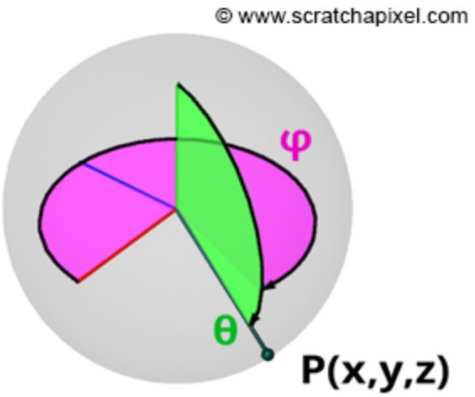
```
vector3 N = normalize(Phit – C)
```



Ray-sphere intersection: texture coordinates

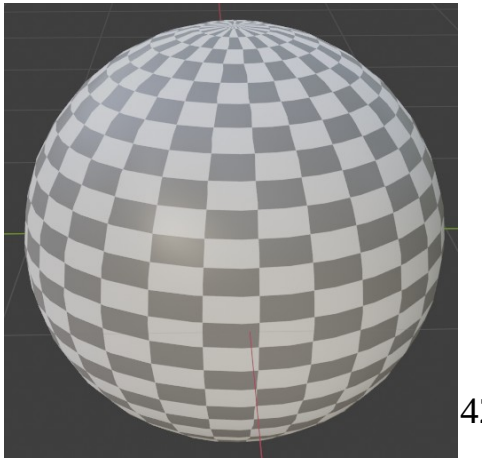
- Sphere can be written in parametric form:

$$P.x = \cos(\theta)\sin(\phi),$$
$$P.y = \cos(\theta),$$
$$P.z = \sin(\theta)\sin(\phi).$$



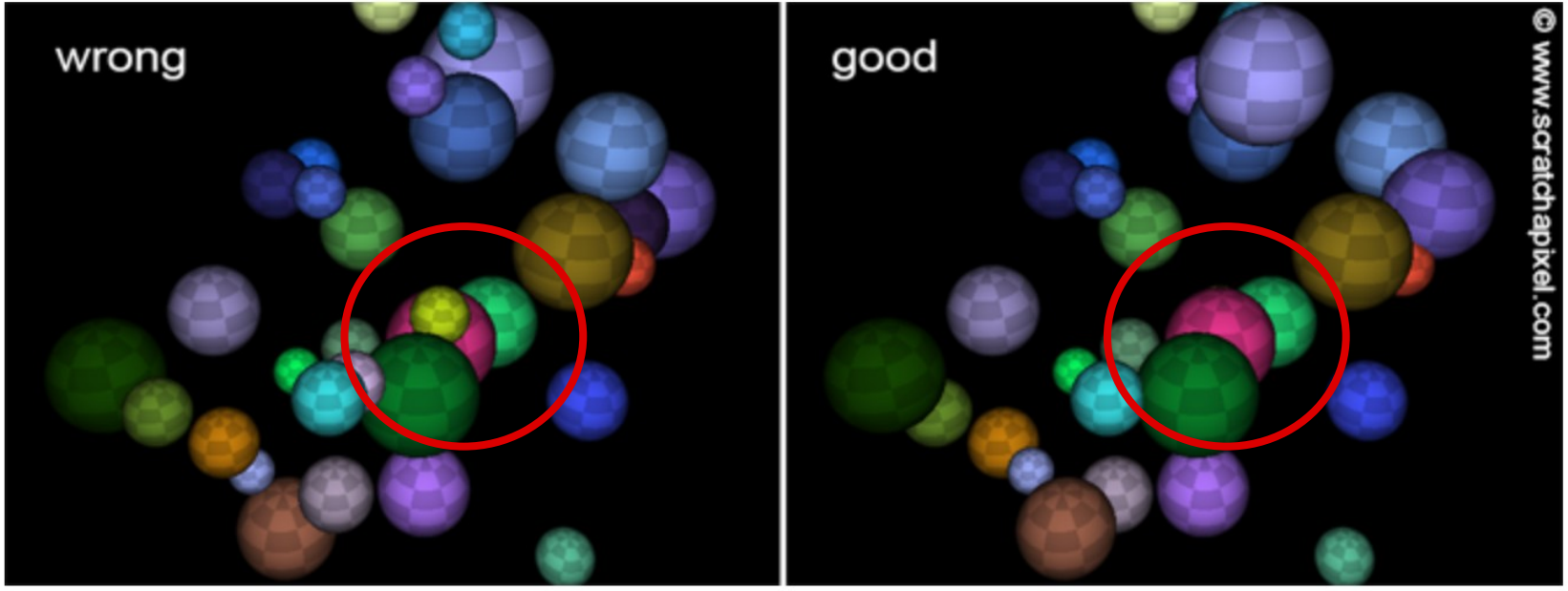
- Texture coordinates for sphere are simply spherical coordinates:

$$\phi = atan(z, x),$$
$$\theta = acos(\frac{P.y}{R}).$$



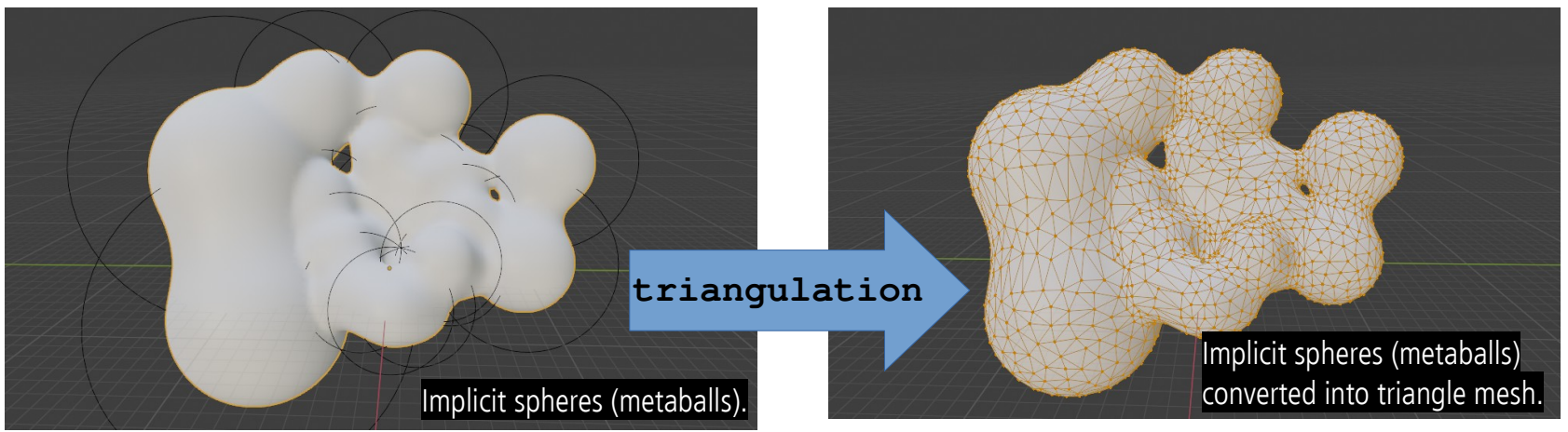
Ray-sphere intersection: intersection order

- If scene contains multiple objects then certain **depth sorting** is needed
  - Keep track of closest intersection distance, closest `t`, and use this value for comparison when testing intersection with new object



Intersecting other shape representations

- Define ray-shape intersection for each representation separately (e.g., triangulated mesh and parametric curves)
- Alternative solution: **convert each shape representation to same internal representation which is used for rendering**
  - Professional rendering software often work on **triangulated mesh** which is obtained using **triangulation**



Tessellation and triangulated mesh

- Two sides of the computer graphics/image generation: authoring of 3D scene and rendering
  - Some representations are much more easier and efficient to handle on authoring level
  - Some are much more efficient to handle for rendering purposes
  - Therefore, efficient mapping of those representations quite important and researched.
- Renderer working with single primitive is much more efficient than supporting various primitives
- Why triangles?
  - Can approximate any surface and shape well
  - Conversion of almost any type of surface to a triangulated mesh (tessellation) is well researched and feasible
  - Triangulated mesh is also basic rendering primitive for rasterization-based renderers
  - Graphics hardware is adapted and optimized to efficiently process triangles
  - Triangles are necessary co-planar which makes various computations, such as ray-triangle, much easier
  - Lot of research was devoted to efficient computation of ray-triangle intersection: https://www.realtimerendering.com/intersections.html
  - For triangles we can easily compute barycentric coordinates which are essential to shading
- Tessellation process can be done after modeling of shape is done and when exporting takes place or it can be done during rendering (render time)

Ray-triangle intersection test

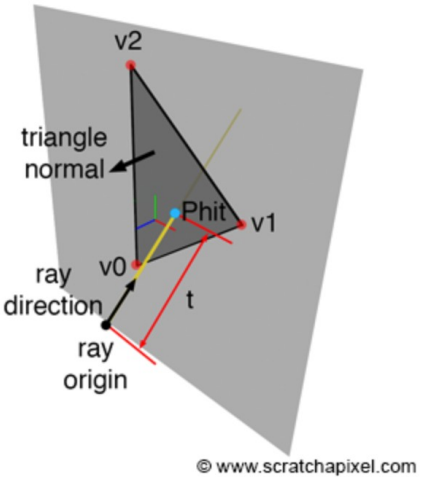
- Basic ray-triangle intersection is straight forward, complexity comes due to multitude of different cases which must be accounted for
  - Thus, there are several algorithms that have been developed and are being developed: https://www.realtimerendering.com/intersections.html
- Ray-triangle intersection testing steps:
  - Does ray intersect a plane defined by a triangle?
  - Does ray intersect point inside the triangle?

Ray-triangle intersection tools

- **Triangles are coplanar**: vertices are lying on a plane and plane can be defined with those vertices
- **Using triangle vertices, normal can be computed**
  - Plane on which triangle lies has the same normal

```
vector3 a = v1 - v0
vector3 b = v2 - v0
vector3 c = cross(a, b)
vector3 normal = normalize(c)
```

  - Winding order of vertices defines normal and thus **surface orientation – important for shading!**



$$\mathbf{a} = a_1\mathbf{i} + a_2\mathbf{j} + a_3\mathbf{k}$$
$$\mathbf{b} = b_1\mathbf{i} + b_2\mathbf{j} + b_3\mathbf{k}$$

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$

$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

https://en.wikipedia.org/wiki/Cross_product

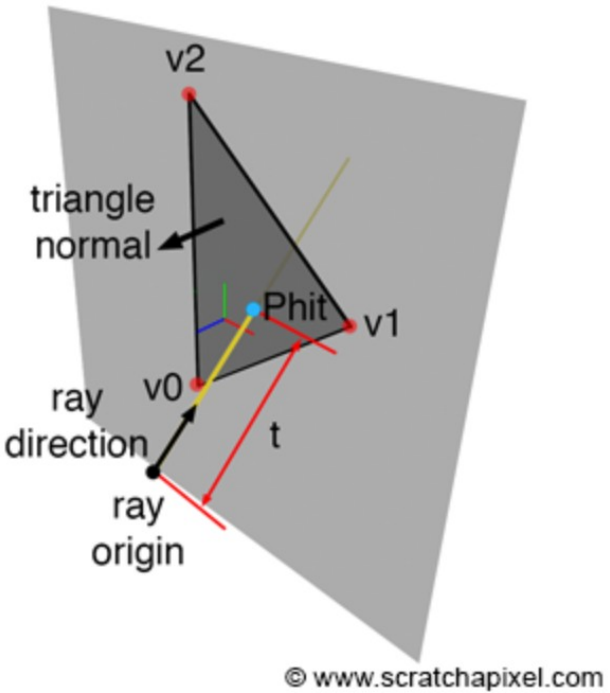
Ray-triangle intersection test: intersecting plane

- Intersected point is somewhere on ray:
  - `Phit = P(t) = O + t * R`
- Plane equation :

  `Ax + By + Cz + D = 0`

  `D = -(Ax + By + Cz)`
  - `A, B, C` are coordinates of plane normal `N = (A,B,C)`
  - `D` is distance from origin `(0,0,0)` to the plane
  - `x, y, z` are coordinates of point on a plane
- Normal `N` can be calculated using triangle vertices
- D can be calculated using any triangle vertex `v` and normal `N`:

  `D = dotProduct(N, v0) = -(N.x * v0.x + N.y * v0.y + N.z * v0.z);`



Ray-triangle intersection: intersecting plane

- Substitute ray equation `P(t) = O + t * R` into plane equation `Ax + By + Cz + D = 0`:

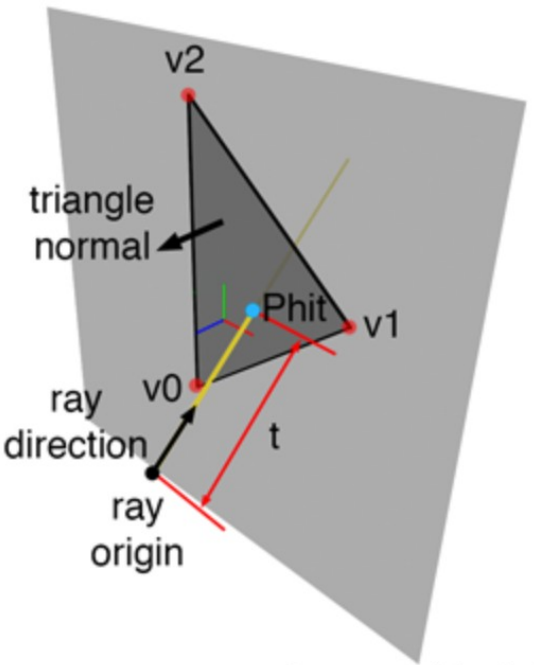
```
A * P.x + B * P.y + C * Pz + D = 0
A * (O.x + t * R.x) + B * (O.y + t * R.y) + C * (O.z + t * R.z) + D = 0
```

- Solving by t:

$$t = -\frac{N(A,B,C) \cdot O + D}{N(A,B,C) \cdot R}$$

- Finally, intersection point of ray and plane:

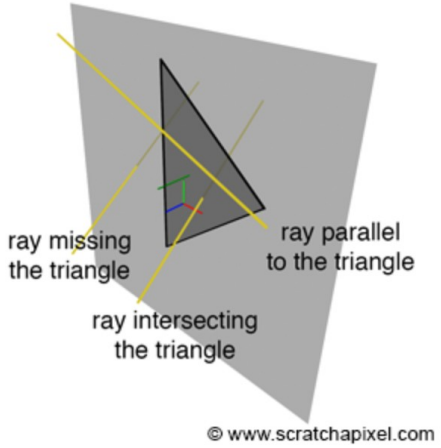
```
float t = - (dot(N, O) + D) / dot(N, R)
Vector3 Phit = O + t * R
```

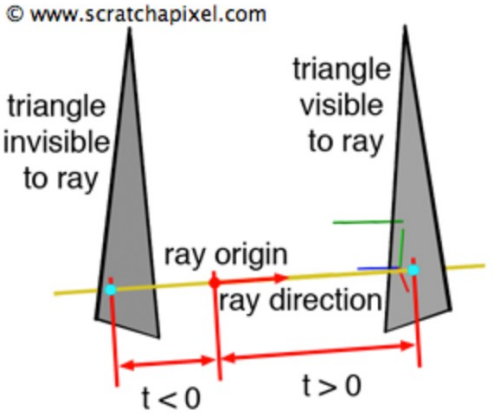


Ray-triangle intersection: intersecting plane

- Special cases of non-intersection:
  - **Ray and triangle (plane) are parallel**
    - Triangle's normal and ray direction are perpendicular

      `dot(N, R) = 0`
  - **Triangle is behind ray origin**
    - If `t < 0` → triangle behind ray origin. Else, triangle is visible

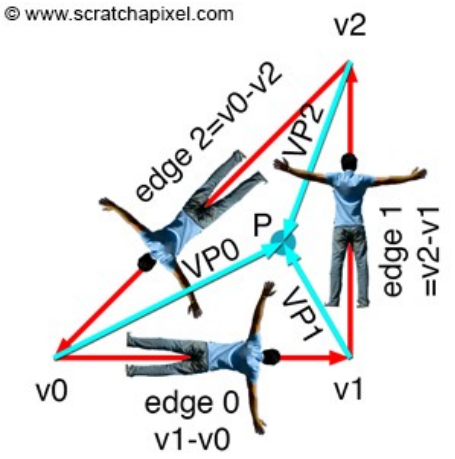




Ray-triangle intersection: point inside triangle?

- We have found intersection point `P` on plane. Is it inside triangle?
- **Inside-out test**

```
vector3 edge0 = v1 – v0;
vector3 edge1 = v2 – v1;
vector3 edge2 = v0 – v2;
vector3 C0 = P – v0;
vector3 C1 = P – v1;
vector3 C2 = P – v2;
bool q1 = dotProduct(N, crossProduct(edge0, C0)) > 0;
bool q2 = dotProduct(N, crossProduct(edge1, C1)) > 0;
bool q3 = dotProduct(N, crossProduct(edge2, C2)) > 0;
If (q1 and q2 and q3) then inside;
```



Inside-outside test:

- Test if dot product of vector along edge and vector defined with first vertex of the test edge and P is positive → if P is on left side of the edge.
- If P is on the left side of all three edges, then P is inside triangle

Ray intersection with triangle mesh

- We have a **routine to compute ray-triangle intersection**
- **To test ray intersection with object which is represented as triangulated mesh:**
  - **Loop over all triangles of triangulated mesh**
    - Test if ray intersects triangles of triangulated mesh
    - Respect depth sorting by keeping track of nearest object
- **For generated camera ray, we can write `intersect()` function which:**
  - Takes triangulated mesh and ray
  - Returns information on intersection
  - Return information on intersection context

https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-polygon-mesh/ray-tracing-polygon-mesh-part-1.html

Testing ray intersections: `intersect()` function

```
bool intersect (Ray ray, Object objects, &intersectionContext)
{
    bool intersected = true;
    for (int k = 0; k < objects.nObjects; ++k)
    {
        for (int n = 0; n < objects[k].nTriangles; ++n)
        {
            if(rayTriangleIntersect(ray, objects[k].triangle[n])
            {
                intersected = true;
                IntersectionContext.objIdx = k;
                IntersectionContext.triIdx = n;
            }
        }
    }
    return intersected;
}
```

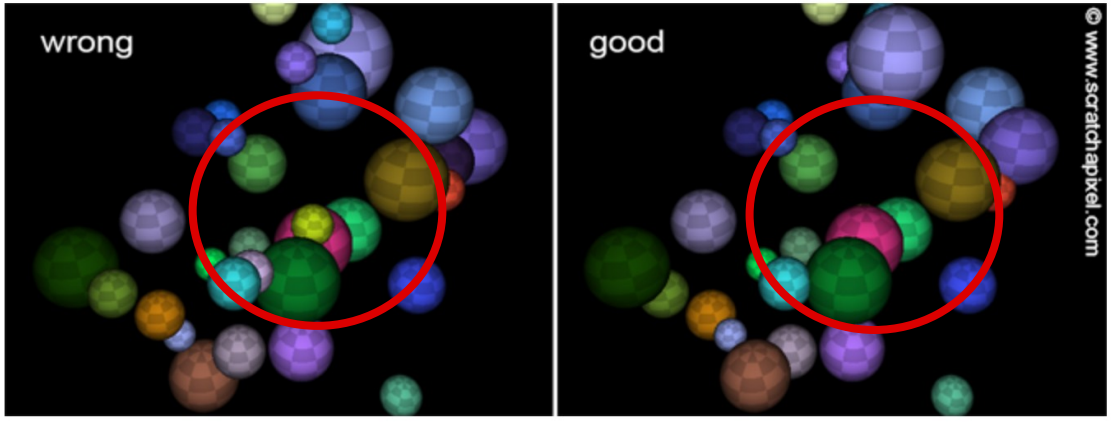
Intersect function can be used for:

- finding intersections between camera rays and objects → **camera visibility**
- Finding intersections between secondary rays and objects → **light transport**

intersect(): depth

```
bool intersect (Ray ray, Object objects, &intersectionContext, &tNearest)
{
    bool intersected = true;
    tnearest = INFINITY;
    for (int k = 0; k < objects.nObjects; ++k)
    {
        for (int n = 0; n < objects[k].nTriangles; ++n)
        {
            if(rayTriangleIntersect(ray, objects[k].triangle[n], t) and t < tNearest)
            {
                intersected = true;
                IntersectionContext.objIdx = k;
                IntersectionContext.triIdx = n;
                tNearest = t;
            }
        }
    }
    Return intersected;
}
```

- Ray may intersect several triangles.
- Keep track of closest intersection and update it with each intersection

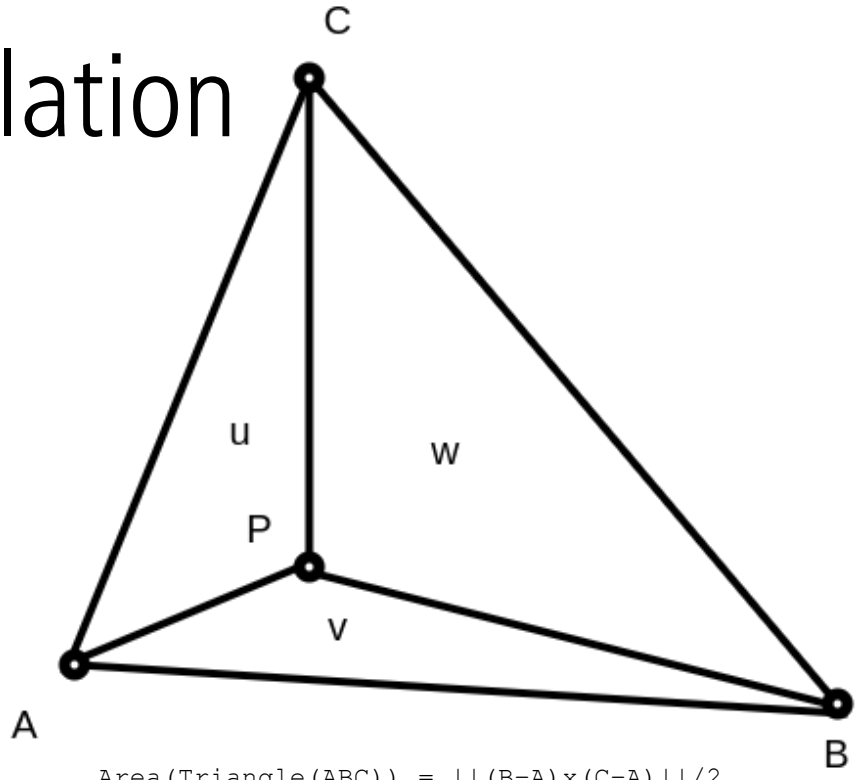


`intersect()` function: intersection context

- `intersect()` function must return information **if ray intersects object** (triangle) as well as **intersection context**:
  - **Object index**: `objIdx`
  - **Triangle index**: `triIdx`
  - **Ray parameter `t` for nearest intersection**: `tNearest`
  - **Intersection point coordinates**
  - **Intersection point normal**
  - **Intersection point texture coordinates**
  - etc.
  - Information of `objIdx`, `triIdx` and `tNearest` can be used to compute the rest of intersection context information
- **Often, triangulated meshes contain information stored per vertices, barycentric interpolation is used to obtain value for specific intersection point**
- **Intersection context information will be used for shading the intersected point**

Recap: barycentric interpolation

- Any point `P` on triangle is described as:
  - `P = uA + vB + wC`, where `A,B,C` are triangle vertices
  - `u,v,w` are **barycentric (areal) coordinates**
    - `u + v + w = 1`
    - When `0 <= u,v,w <=1` `P` inside or on triangle edge. Otherwise P is outside of triangle.
- Barycentric coordinates: `u,v,w` are proportional to area of sub-triangles defined by P
- Interpolating vertex data
  - We know `P` and `A,B,C`. Calculate `u,v,w` and use this factors for interpolating



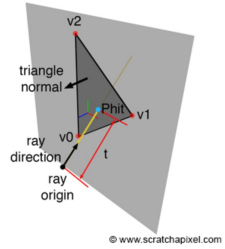

```
Area(Triangle(ABC)) = ||(B-A)x(C-A)||/2

u = Area(Triangle(CAP)) / Area(Triangle(ABC))
v = Area(Triangle(ABP)) / Area(Triangle(ABC))
w = Area(Triangle(BCP)) / Area(Triangle(ABC))
```

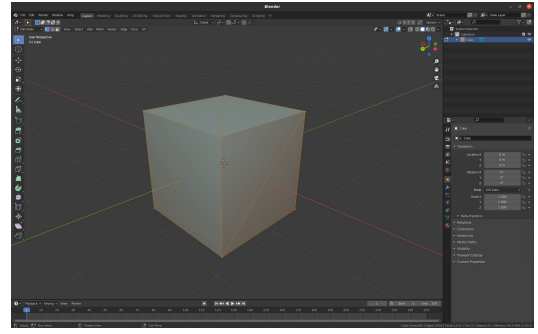
Intersection context: normal

- Triangle normal, thus intersection point normal, can be calculated from triangle vertices:

```
vector3 a = v1 - v0
vector3 b = v2 - v0
vector3 c = cross(a, b)
vector3 normal = normalize(c)
```



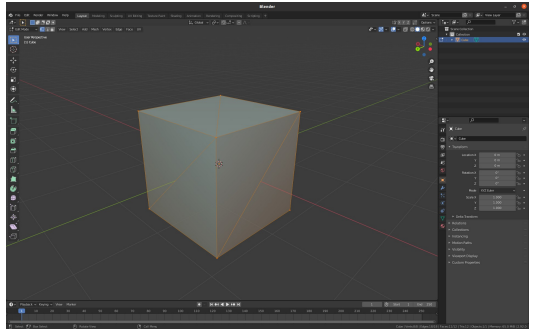
- Smooth normals can be created when modeling triangulated mesh
  - Those are stored per vertex
  - To obtain smooth normal in intersection point, barycentric interpolation can be performed using intersection point and triangle vertices



```
# Blender v2.92.0 OBJ File: ''
# www.blender.org
o Cube_Cube.002
v -1.000000 -1.000000 1.000000
v -1.000000 1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v -1.000000 1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v 1.000000 1.000000 1.000000
v 1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -1.000000
vt 0.625000 0.000000
vt 0.375000 0.250000
vt 0.375000 0.000000
vt 0.625000 0.250000
vt 0.375000 0.500000
vt 0.625000 0.500000
vt 0.375000 0.750000
vt 0.625000 0.750000
vt 0.375000 1.000000
vt 0.125000 0.750000
vt 0.125000 0.500000
vt 0.875000 0.500000
vt 0.625000 1.000000
vt 0.875000 0.750000
vn -1.0000 0.0000 0.0000
vn 0.0000 0.0000 -1.0000
vn 1.0000 0.0000 0.0000
vn 0.0000 0.0000 1.0000
vn 0.0000 -1.0000 0.0000
vn 0.0000 1.0000 0.0000
s off
f 2/1/1 3/2/1 1/3/1
f 4/4/2 7/5/2 3/2/2
f 8/6/3 5/7/3 7/5/3
f 6/8/4 1/9/4 5/7/4
f 7/5/5 1/10/5 3/11/5
f 4/12/6 6/8/6 8/6/6
f 2/1/1 4/4/1 3/2/1
f 4/4/2 8/6/2 7/5/2
f 8/6/3 6/8/3 5/7/3
f 6/8/4 2/13/4 1/9/4
f 7/5/5 5/7/5 1/10/5
f 4/12/6 2/14/6 6/8/6
```

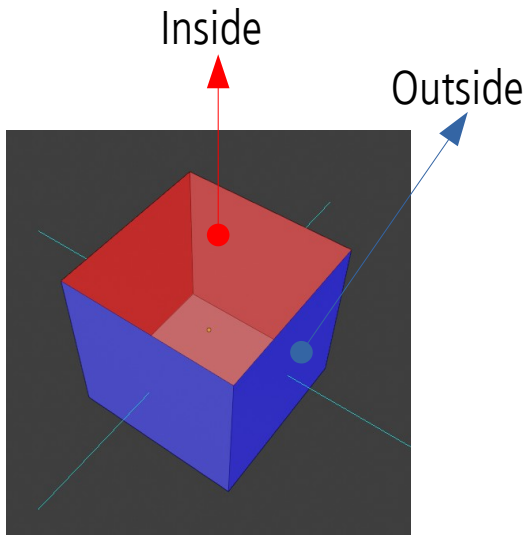
Intersection context: texture coordinates

- Texture coordinates for each vertex of triangulated meshes are created during mesh modeling using:
  - **Mesh unwrapping**
  - **Texture projections**, e.g., spherical, cylindrical, triplanar, etc.
- To compute texture coordinate in intersection point, barycentric interpolation is used
- Note: texture projections can be also used to create texture coordinates on the fly

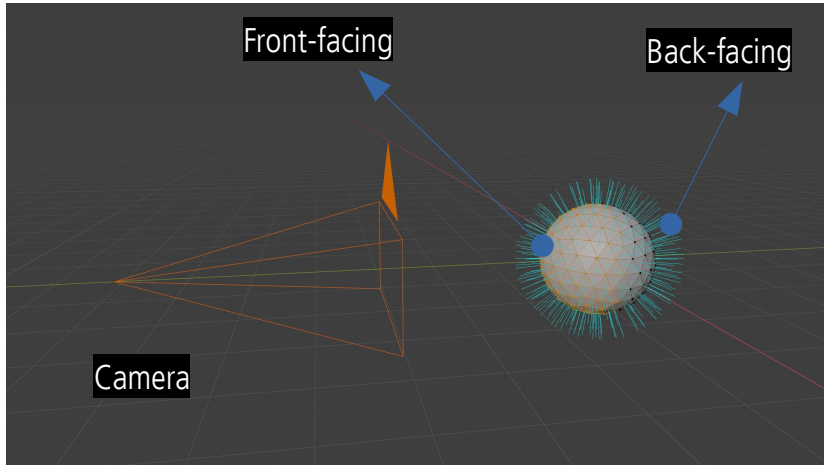


```
# Blender v2.92.0 OBJ File: ''
# www.blender.org
o Cube_Cube.002
v -1.000000 -1.000000 1.000000
v -1.000000 1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v -1.000000 1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v 1.000000 1.000000 1.000000
v 1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -1.000000
vt 0.625000 0.000000
vt 0.375000 0.250000
vt 0.375000 0.000000
vt 0.625000 0.250000
vt 0.375000 0.500000
vt 0.625000 0.500000
vt 0.375000 0.750000
vt 0.625000 0.750000
vt 0.375000 1.000000
vt 0.125000 0.750000
vt 0.125000 0.500000
vt 0.875000 0.500000
vt 0.625000 1.000000
vt 0.875000 0.750000
vn -1.0000 0.0000 0.0000
vn 0.0000 0.0000 -1.0000
vn 1.0000 0.0000 0.0000
vn 0.0000 0.0000 1.0000
vn 0.0000 -1.0000 0.0000
vn 0.0000 1.0000 0.0000
s off
f 2/1/1 3/2/1 1/3/1
f 4/4/2 7/5/2 3/2/2
f 8/6/3 5/7/3 7/5/3
f 6/8/4 1/9/4 5/7/4
f 7/5/5 1/10/5 3/11/5
f 4/12/6 6/8/6 8/6/6
f 2/1/1 4/4/1 3/2/1
f 4/4/2 8/6/2 7/5/2
f 8/6/3 6/8/3 5/7/3
f 6/8/4 2/13/4 1/9/4
f 7/5/5 5/7/5 1/10/5
f 4/12/6 2/14/6 6/8/6
```

Single and double sided triangle



- **Winding order of triangle vertices and handedness of coordinate system defines normal orientation** → surface orientation
  - Surface normal pointing outward → **outside surface**
  - Surface normal pointing inward → **inside surface**
- Based on camera view, surface can be:
  - **Front-facing**: if outside surface is facing camera
  - **Back-facing**: if outside surface is not facing camera
- **During rendering, that is visibility computation**, it is possible to specify:
  - **Single-sided primitives**: only visible if are front-facing
  - **Double sided primitives**: visible for both front- and back-facing
- **Back-face culling**: discarding back-facing triangles during rendering
  - For casting shadows, back-face culling can not be used.
  - Test: `dotProduct(ray.direction, N) > 0`

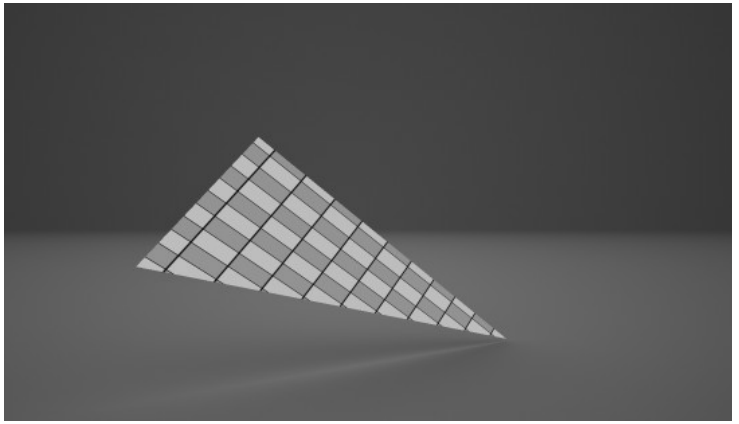


Testing intersections: complexity

- **Complexity of ray-tracing algorithm depends on:**
  - **Number of pixels of virtual image plane** → number of camera rays
    - Note that often multiple rays per pixel are used, e.g., 256
  - **Number of objects in the scene**
- Each camera ray must be tested for intersection with every object
- Every secondary and shadow ray must be tested for intersection with every object
  - Additional rays are generated per secondary ray intersections → recursion

Testing intersections: efficiency

- Time to render a scene is (directly) proportional to the number of triangles in the scene
  - For every camera ray, looping over all objects, that is triangles is needed
- For shading purposes, all triangles must be stored into memory and each must be tested for intersection for ray
- Therefore, various efficient ray-triangle and other ray-object intersection tests exist:
  - Möller-Trumbore method: https://www.tandfonline.com/doi/abs/10.1080/10867651.1997.10487468
  - Philip Dutré and Ares Lagae: https://www.tandfonline.com/doi/abs/10.1080/2151237X.2005.10129208
  - Marta Löfstedt and Tomas Akenine-Möller: https://www.tandfonline.com/doi/abs/10.1080/2151237X.2005.10129195
  - Inigo Quilez: https://www.shadertoy.com/view/MlGcDz
  - More: https://www.realtimerendering.com/intersections.html



Moana Island

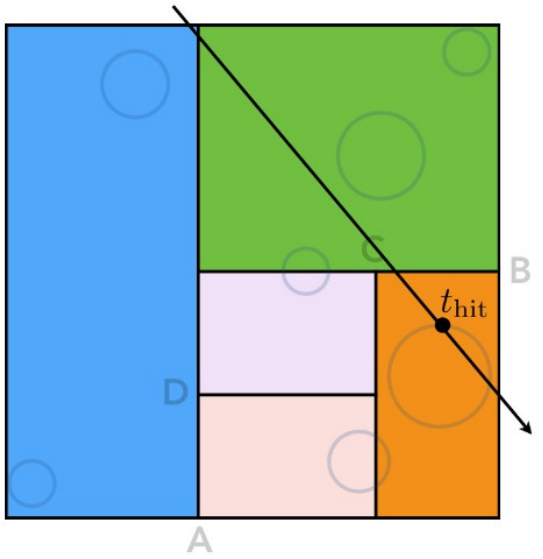
>90 million unique quads and triangles
>5milion curves
>28milion instances of objects
→ 15 billion primitives

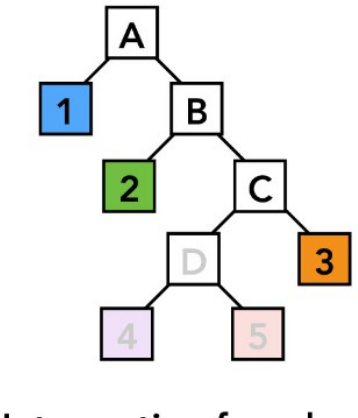
https://disneyanimation.com/resources/moana-island-scene/



Accelerating intersections

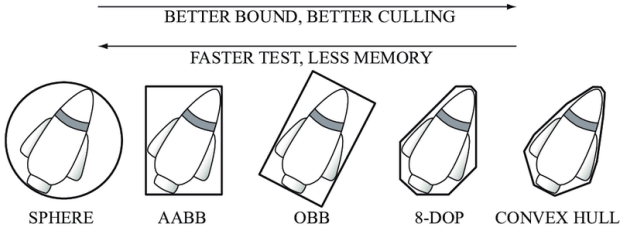
- If certain parts of the scene are not relevant for intersecting with current ray, they can be skipped
- **Spatial acceleration data structures**
  - 3D scene and scene is spatially subdivided
  - Now larger volumes of the scene can be tested for intersection
  - If ray is not intersecting volume, all objects inside do not have to be tested for intersections
- Acceleration structures require **pre-computation overhead**:
  - **Static scenes**: only once before rendering
  - **Animated scenes**: each frame, each time objects move



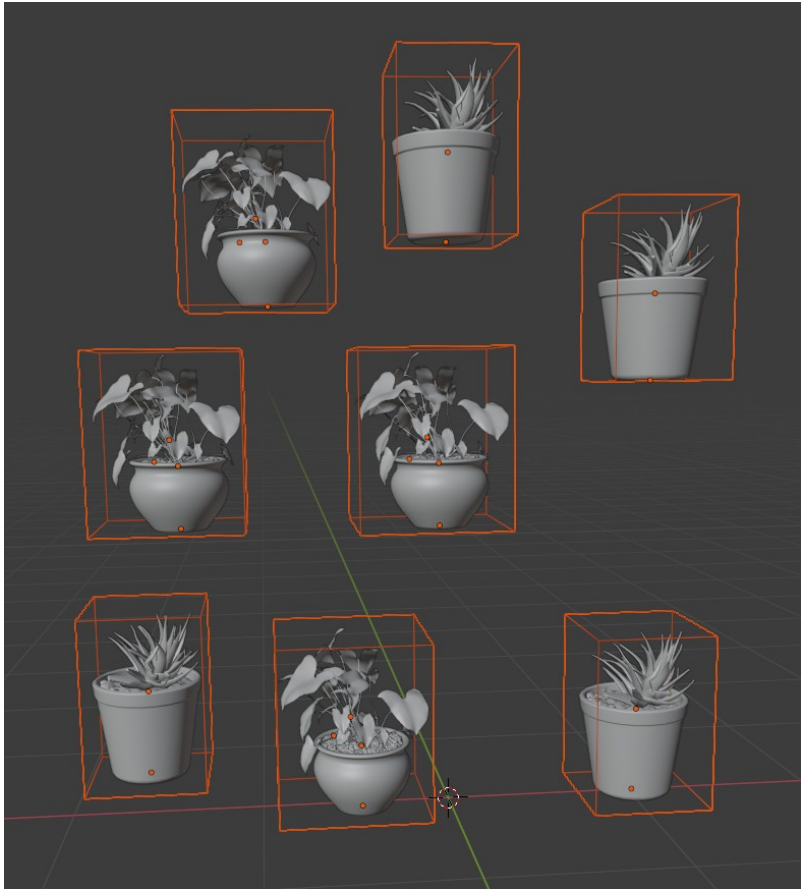


https://cs184.eecs.berkeley.edu/sp22/lecture/10/ray-tracing-acceleration

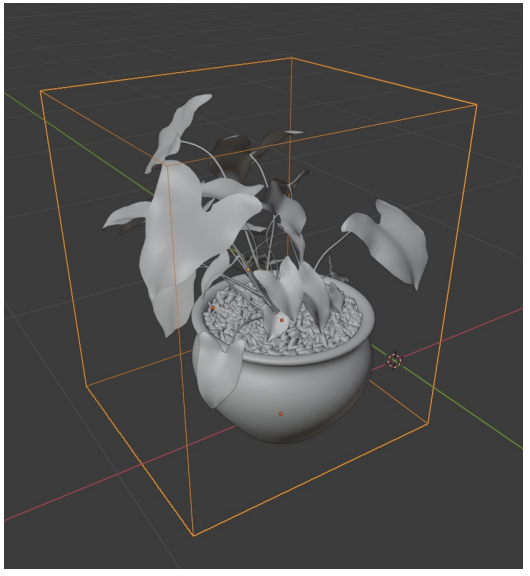
Bounding volume



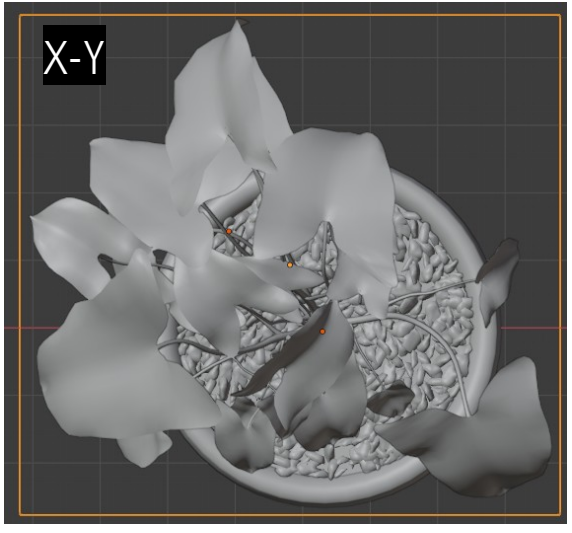
- Bounding volume: tightest possible volume (e.g., box or sphere) surrounding object
- Idea behind bounding volume:
  - For all objects in 3D scene, first test bounding volume intersection with ray
    - If ray doesn't intersects volume, then skip complete object inside
    - If ray intersects volume, loop over object triangles and test for intersection
- Testing intersection between ray and bounding volume is simple:
  - Sphere bounding volume: ray-sphere intersection
  - Box bounding volume: box is triangulated mesh → ray-triangle intersection



Bounding box

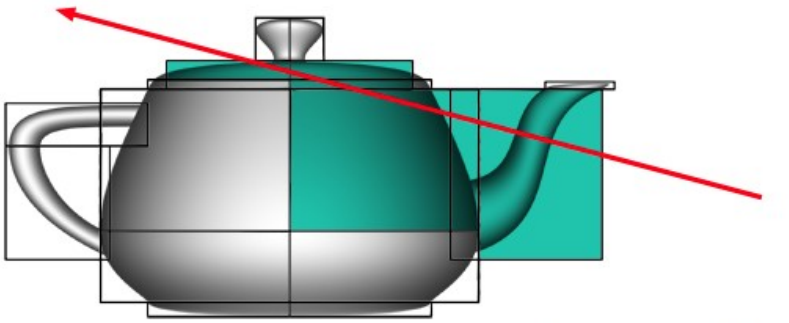


- **Building bounding box for any object is simple:**
  - Loop over all vertices of object mesh and find minimum and maximum value for each vertex (x,y,z)
  - Found minimum and maximum coordinates are bounding box corners
- Building bounding box can be done once only once and reused

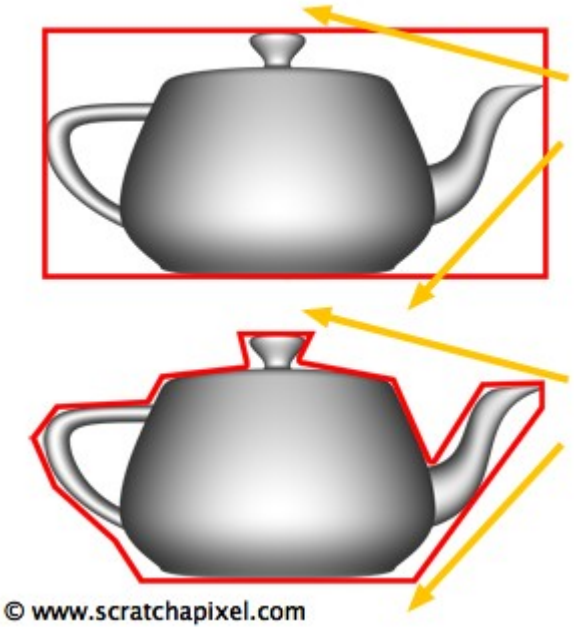


Bounding volume optimization

- One bounding volume (e.g., box) may fit object too loosely
  - Ray that intersect bounding box will often miss object
- Bounding extent can be created out of multiple bounding volumes to fit objects more closely
  - Gives better results but it is more costly to ray-trace



© www.scratchapixel.com



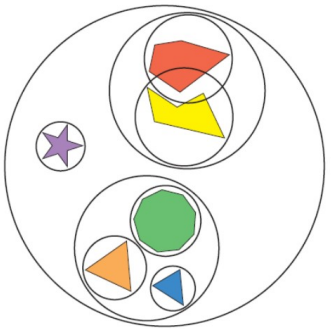
© www.scratchapixel.com

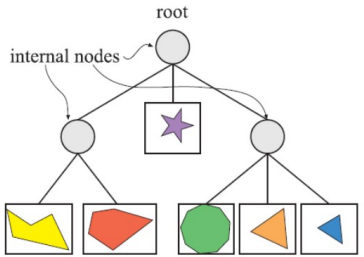
Spatial data structures

- **Organizes objects in 3D scene using hierarchical, tree-like structure**
  - Root defines whole scene
  - Children nodes define its own volume of space which in turn contains its own children
- **Structure is nested and recursive**
- For n objects it gives **improvement from `O(n)` to `O(log(n))`** for testing intersections
- **Main types:**
  - **Bounding volume hierarchies** → object subdivisions
  - **Binary space partitioning trees** → space subdivision

Bounding volume hierarchies (BVH)

- Encloses the regions of space surrounding objects
- **Bounding volumes are much simpler to intersect than the contained objects**
- **Examples of bounding volumes:**
  - Spheres
  - Axis-aligned bounding boxes (AABB)
  - Oriented bounding boxes (OBB)
- **Scene is organized into a hierarchical tree structure consisting of a set of connected nodes**
  - Root node contains whole scene
  - Internal nodes contain intermediate volumes and point to other internal nodes with smaller volumes
  - Leaf nodes are volumes containing actual object geometries

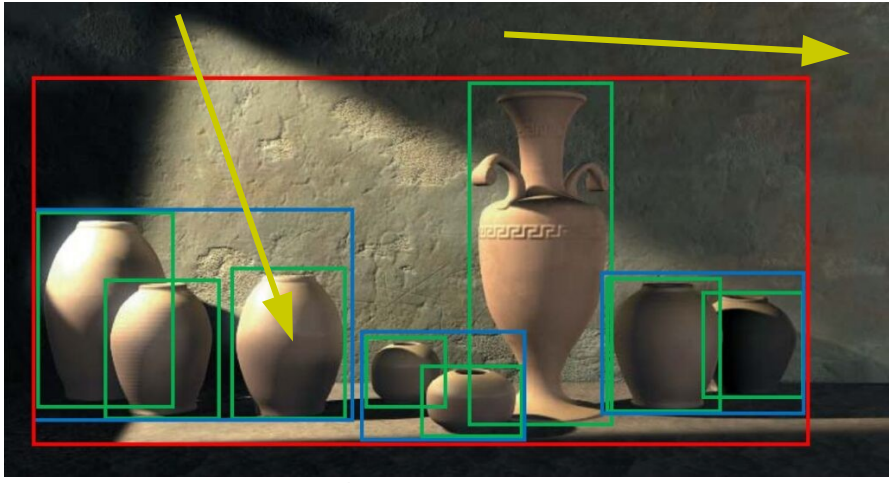




https://courses.engr.illinois.edu/cs419/sp2017/Lecture-19-BVH%20and%20Octrees.pdf

Bounding volume hierarchies

- **Once BVH is constructed it can be used for intersection testing given ray**
  - Testing starts at the root
  - If ray misses volume defined by root then skip testing for the rest of the BVH
  - Otherwise, testing continues recursively by testing the children (internal) nodes.
    - Any time when ray misses volume defined by child node the whole sub-tree can be discarded
  - When ray hits leaf node, ray is tested for intersection with actual object geometry
- **For dynamic (animated) scenes, BVH must be recomputed at each frame when objects move**
  - Example: temporal bounding volume: https://ieeexplore.ieee.org/document/1274066



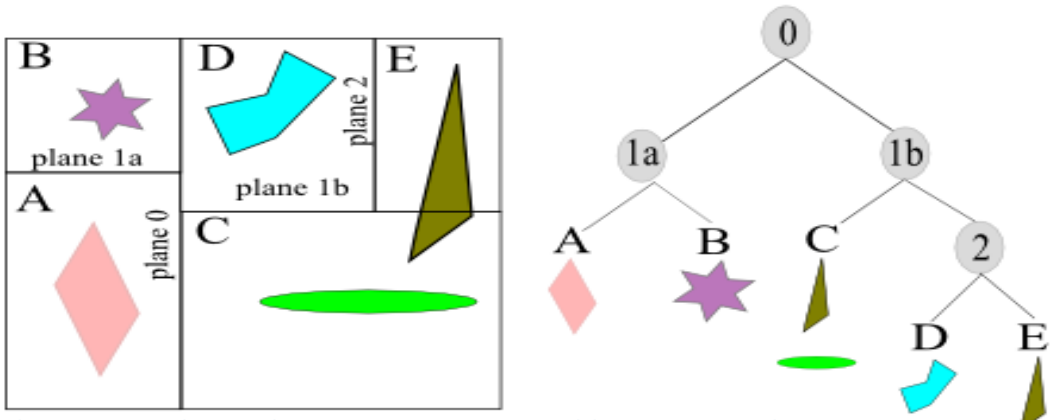
Ray intersection testing with BVH:
https://jacco.ompf2.com/2022/04/13/how-to-build-a-bvh-part-1-basics/

Binary space partitioning (BSP) trees

- Entire space is subdivided and encoded into tree data structure
- **Tree is created by plane subdividing space in two, sorting geometry into these two spaces. Further division is done recursively.**
- Main types:
  - Axis aligned BSP AKA kd trees
  - Octrees
  - Polygon aligned BSP

Axis-aligned BSP construction

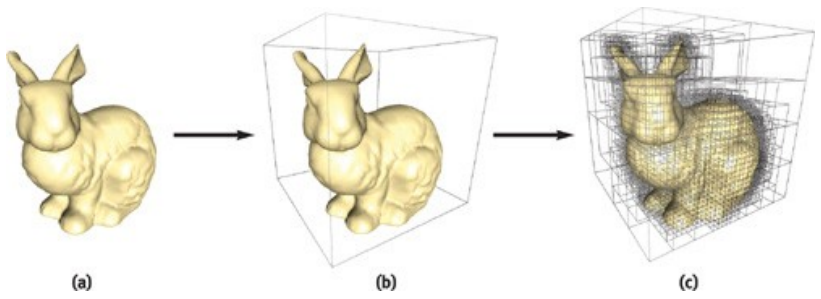
- Fist, whole scene is enclosed in axis-aligned bounding box (AABB)
- Then, initial AABB is recursively subdivided into smaller boxes
- At any level of recursion: one axis of box is choosen and perpendicular plane is used to divide space into two boxes
  - Uniform: plane is positioned exactly at half of the box
  - Non-uniform: plane is positioned adaptively inside box → more balanced tree
- Objects intersecting the planes:
  - Truly split object using plane into two separate objects
    - Only once copy of object, deletion is easier
    - Inefficient for smaller objects
  - Create duplicate and place at both nodes
    - Can give tighter bounds to larger objects
    - Problem of duplicating object

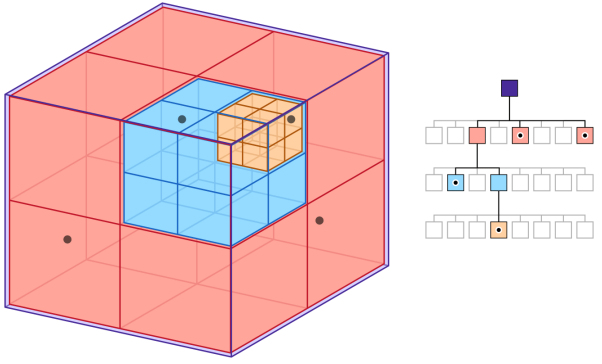


https://www.bogotobogo.com/Games/spatialdatastructure.php

Octrees

- First, scene is enclosed in a minimal axis-aligned (AA) box
- Similarly, to AA-BSP the box is then split simultaneously along three axes recursively where split point is the center of the box
  - Result is eight new boxes, which can be described with tree-like structure: octree
  - Recursive splitting stops when maximum depth is achieved or when there are certain number of objects (triangles) in a box





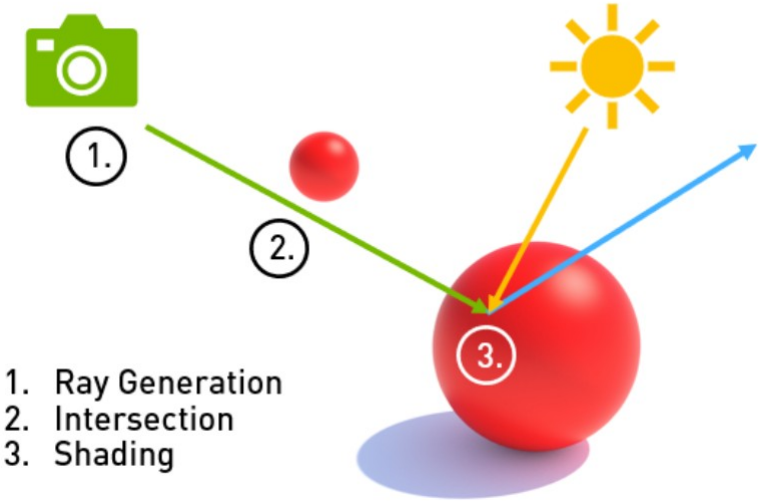
https://developer.nvidia.com/gpugems/gpugems2/part-v-image-oriented-computing/chapter-37-octree-textures-gpu
https://developer.apple.com/documentation/gameplaykit/gkoctree

PLAN

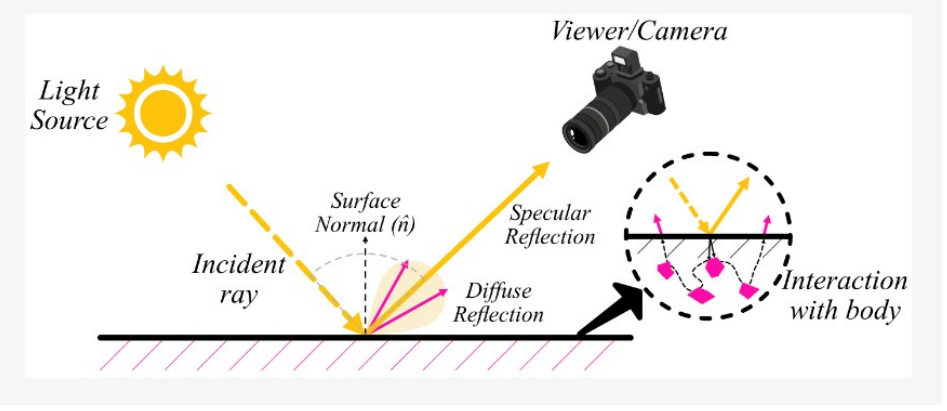
- From here on:
  - Say that we are working with surface rendering (volumetric rendering is out of scope)
  - Discuss rendering equation and say that this is solved by shading
  - Introduce Light transport as a mean for computing incoming light
  - Say that there are different light transport methods: directional, Whitted, global
    - Then focus on direct illumination
    - Focus on Whitted illuminiation
    - Give few comparisons with global illumination: path tracing
  - Finally, discuss materials which determined amount of light reflected in eye: BRDF and texture

Shading

- Once **intersection of camera ray with object** is found (visibility is solved), we need to calculate color and intensity of that point → **shading**
  - Intersection point → **shading point**
- **Shading and thus appearance of object depends on:**
  - **Material of the object surface (scattering model, texture)**
  - **Shape of the object surface (normal)**
  - **Incoming light on surface (direction, color, intensity)**
  - **Camera viewpoint (position and orientation)**
- Color of an object at any point is the result of the way object reflects light falling on that point to camera.



https://developer.nvidia.com/blog/rtx-best-practices/



https://www.mdpi.com/1424-8220/22/17/6552

Rendering equation

- Light (color) in shading point towards view direction is described with **rendering equation**
  - **Light** L_o in **intersected (shading) point** `p` towards **view direction** ω_o is results of summing all **incoming light** L_i from hemisphere of directions Ω, multiplying it with **BRDF** `f(p, ωo, ωi)` and **attenuation factor** `max(0, dotProduct(n, ωi))`

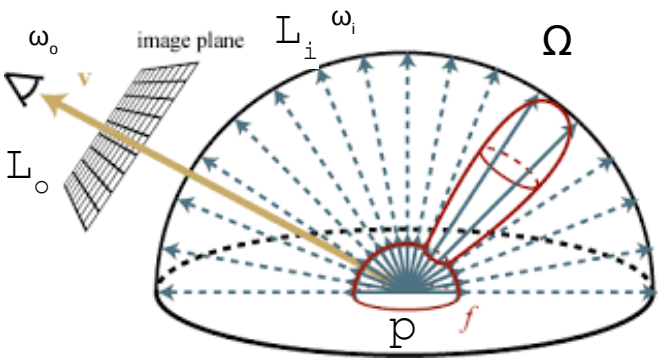
$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_\Omega f(p,\omega_o,\omega_i) L_i(p,\omega_i)(\omega_i \cdot n) d\omega_i$$

Emission
- Take in account contribution of light if surface is emissive

BRDF
- Defines surface material
- Uses texture information

Incoming light over hemisphere of directions

Attenuation due to orientation of surface towards light.
- Depends on surface shape (normal)



Rendering equation decomposition

- **BRDF**, e.g., specular and diffuse surfaces
- **Attenuation factor**: alignment of surface normal and incoming light direction
- **Incoming light**: recursive and describes complex light transport → not possible to solve analytically → **different light transport methods**
  - Solve using assumptions:
    - Only take in account light coming from light sources: **direct illumination**, e.g., point and directional lights
    - Only take in account specular and diffuse surfaces: **Whitted ray-tracing**
  - Solve using approximations: global illumination
    - Monte-Carlo ray-tracing: path-tracing
    - Finite elements methods: radiosity

$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_\Omega f(p,\omega_o,\omega_i) L_i(p,\omega_i) (\omega_i \cdot n) d\omega_i$$

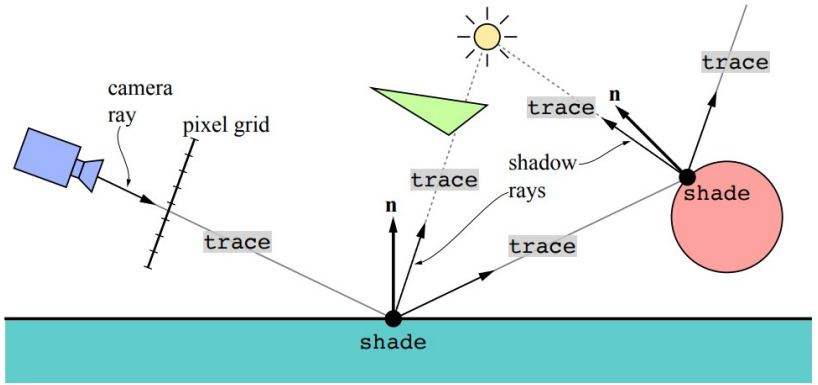
Rendering equation: assumptions

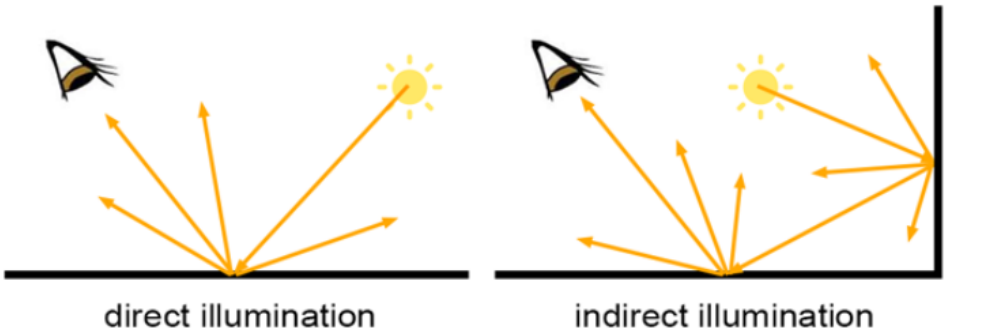
- **Assume: between objects is vacuum**
- **No participating media**: fog, smoke, clouds → **volumetic rendering equation**
  - Light can not get attenuated while traveling between objects
- **We will describe surface reflection**
- **No sub-surface scattering and transmission** → further extensions on rendering equation
  - Light will not scatter inside objects

https://www.pbr-book.org/3ed-2018/Light_Transport_I_Surface_Reflection

Incoming light

- Computing incoming light on surface is essential to shading
  - **Direct illumination**: compute contribution directly and only from light sources
  - **Indirect illumination:** light reflected from another object surface





https://www.realtimerendering.com/Real-Time_Rendering_4th-Real-Time_Ray_Tracing.pdf

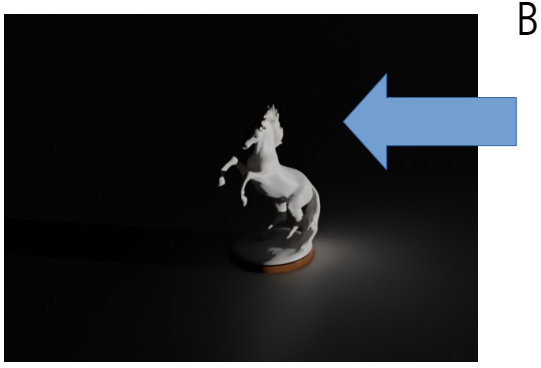
Direct illumination

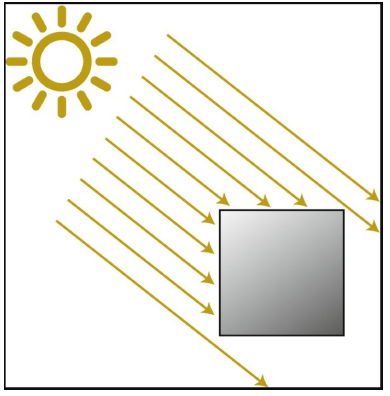
$$c_{shaded(p,v)} = \sum_{i=1}^{n} f(l_i, n, v) c_{light_i} (n \cdot l_i)^+$$

Light source color

BRDF

- **Loop over all light sources and evaluate surface color**
- **Directional lights**
  - Light direction: direction of directional lights can be used directly
  - Light color and intensity are given are parameters





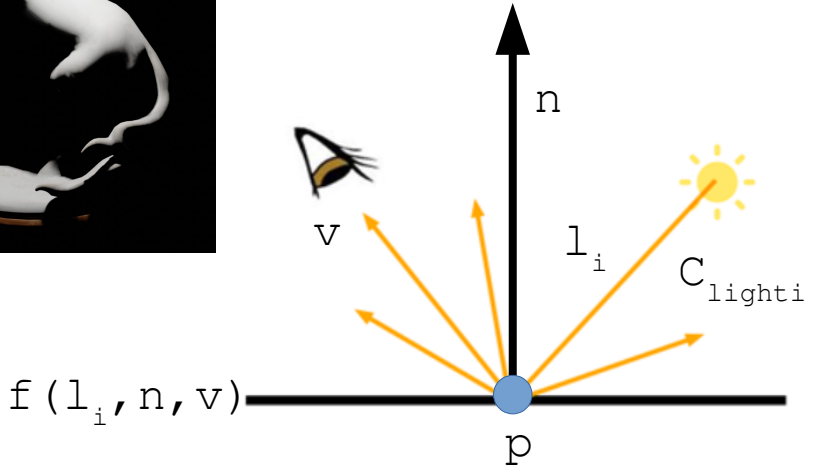
Direct illumination

$$c_{shaded(p,v)} = \sum_{i=1}^{n} f(l_i, n, v) c_{light_i} (n \cdot l_i)^+$$

Light source color

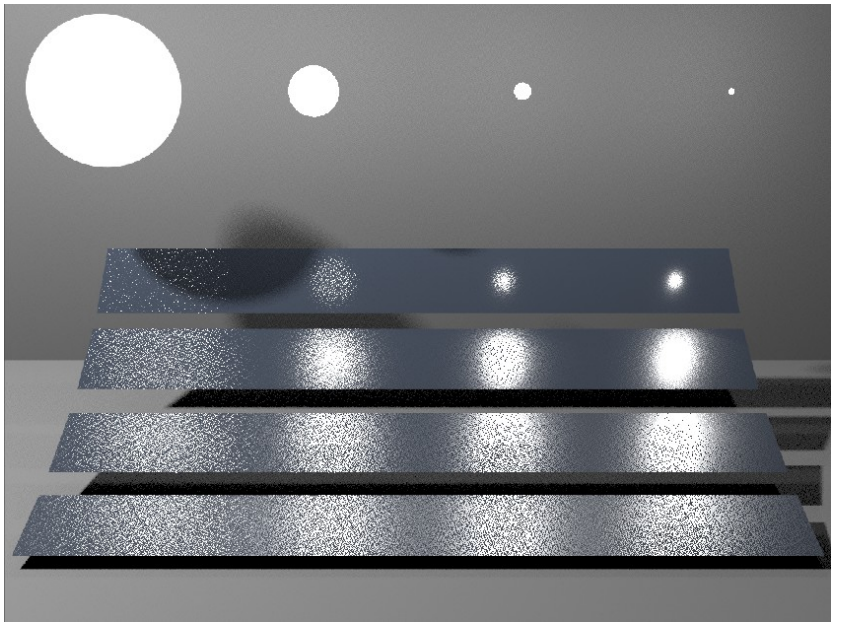
BRDF

- Loop over all light sources and evaluate surface color
- Point lights or spot lights
  - Ray is generated in from shading point **`P(x,y,z)`** to light source **`S(x,y,z)`** (visibility solving):
    - Light direction: `L = normalize(S - P);`
    - Light color and intensity
      - Inverse square law: intensity falls with squared distance to light source



Direct illumination

- Loop over all light sources and evaluate surface color
- Area lights
  - Sample points on light source geometry (Monte-Carlo methods)
  - For each sampled point on light source geometry:
    - Generate ray from shading point to sampled point
    - Evaluate contribution
    - Evaluate surface color



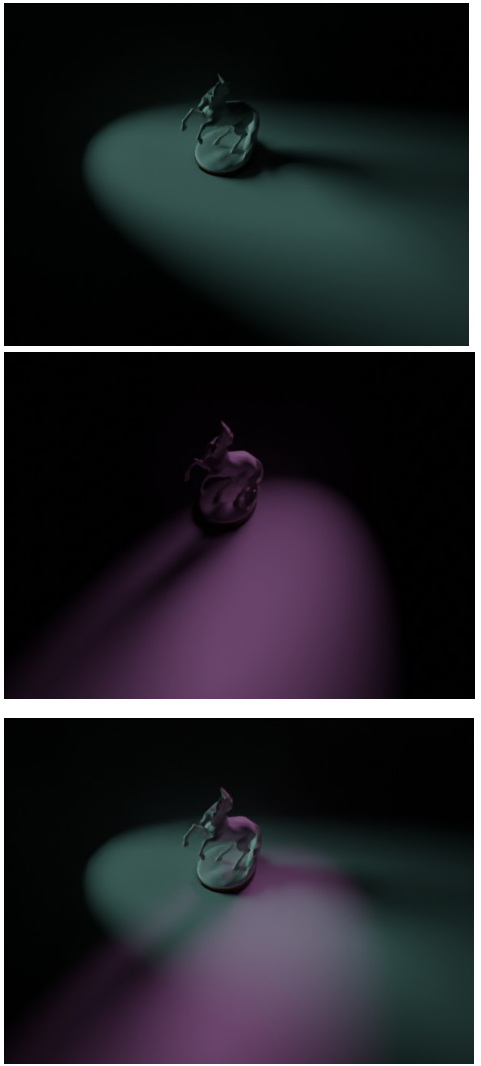
Sampling light sources.
Different size of area light sources.
Different roughness of metal plates (top: very smooth, bottom: very rough)

https://www.pbr-book.org/3ed-2018/Light_Transport_I_Surface_Reflection/Direct_Lighting
https://www.pbr-book.org/3ed-2018/Light_Transport_I_Surface_Reflection/Sampling_Light_Sources

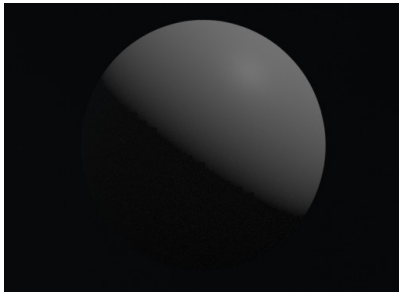
Shading: Multiple lights

- 3D scene often contain multiple light sources
- Contribution of each light source adds up linearly

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_\Omega f(p, \omega_o, \omega_i) L_i(p, \omega_i)(\omega_i \cdot n) d\omega_i$$



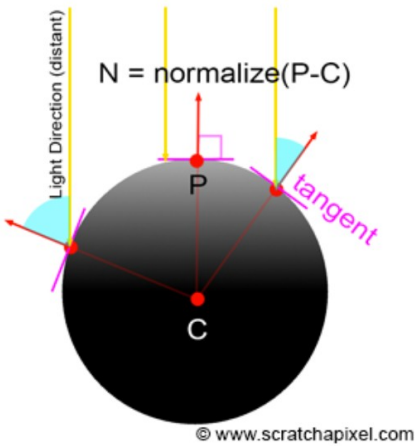
Light attenuation



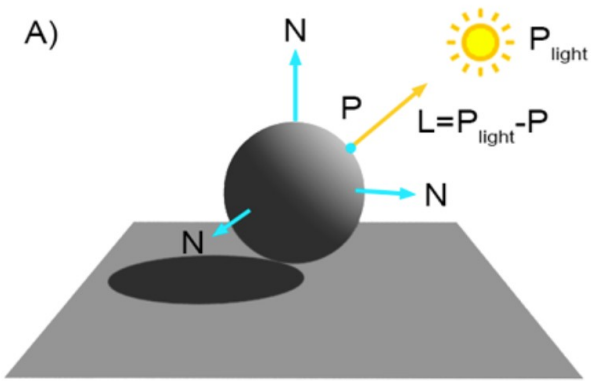
- Surface normal `N` in shading point defines surface orientation
- Alignment of surface normal `(N)` and incoming light direction `(L)` determines surface brightness:

```
max(0, dotProduct(N, L))
```

$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_\Omega f(p,\omega_o,\omega_i) L_i(p,\omega_i)(\omega_i \cdot n) d\omega_i$$



Light attenuation in case of directional lights

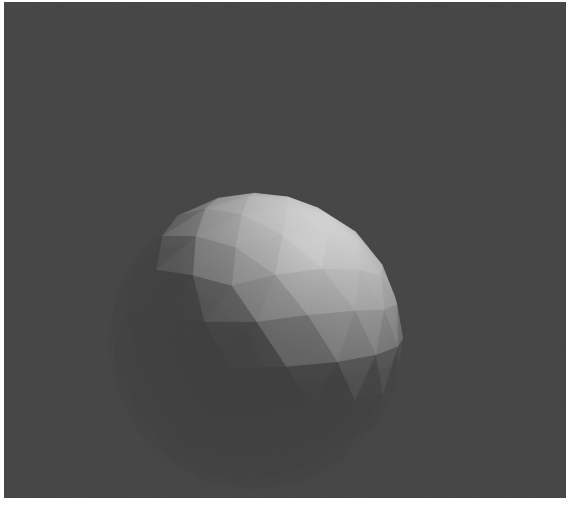


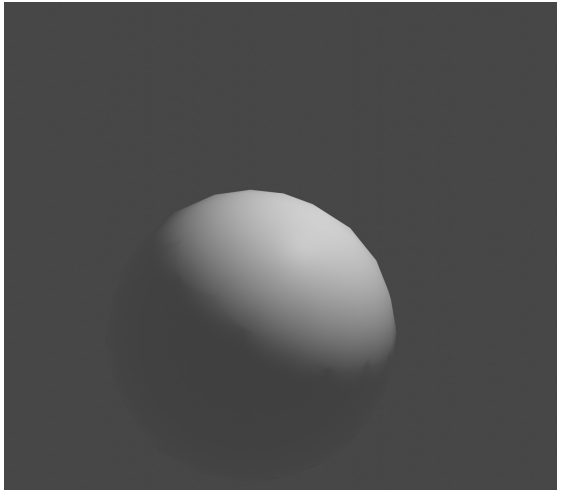
Light attenuation in case of point lights

Incoming light (from any direction or lights source) can get attenuated due to surface orientation

Surface normal

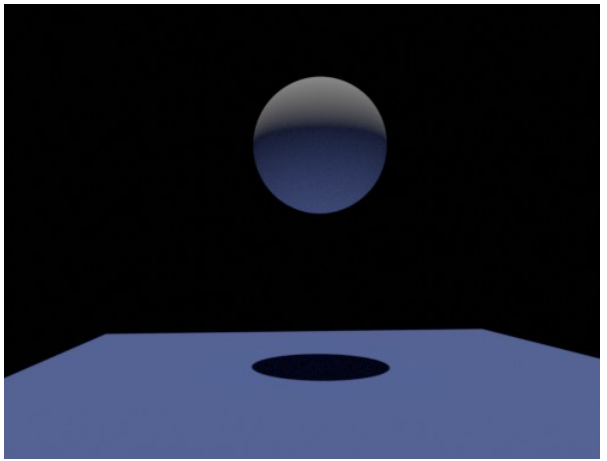
- Triangulated mesh can't represent perfectly smooth surface
- Since shading depends on surface normal, triangulated meshes result in faceted look
  - **Flat shading**
- Solution: Gourad or Phong shading
  - **Smooth shading**

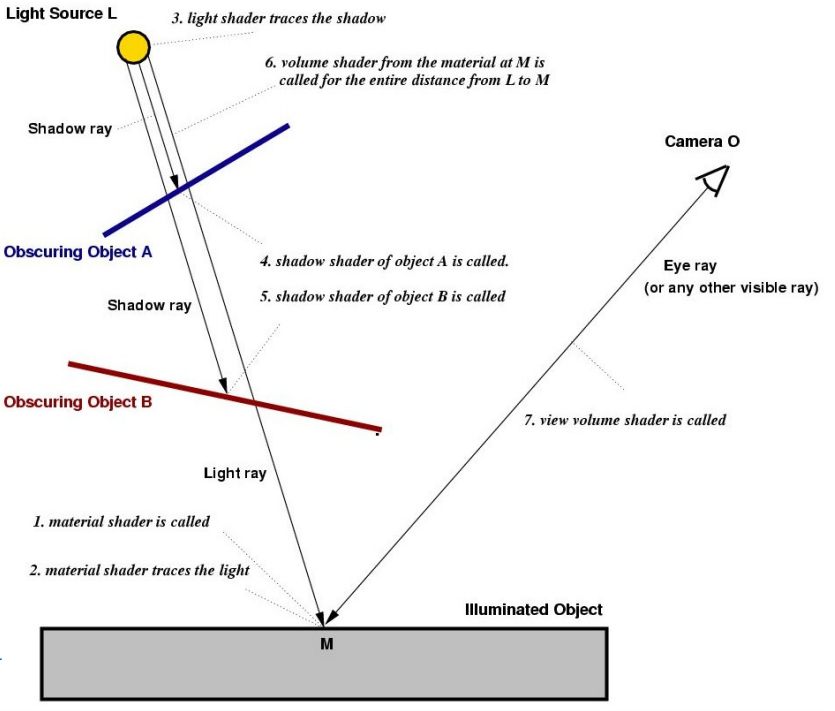




Shadows

- While evaluating incoming light, rays are traced from shading point to light sources (or other surfaces) → **shadow/secondary/light rays**
- Next to light direction, intensity and color, the purpose of shadow ray is to determine if it is obstructed by objects
- If shadow ray is obstructed, it has zero light contribution → black color



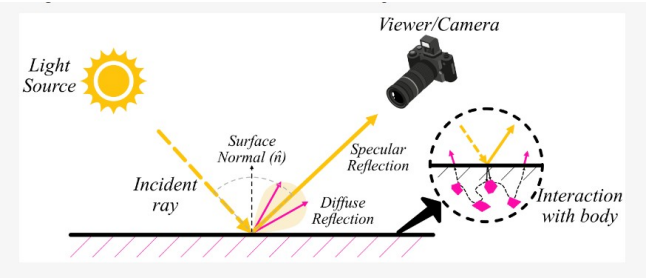


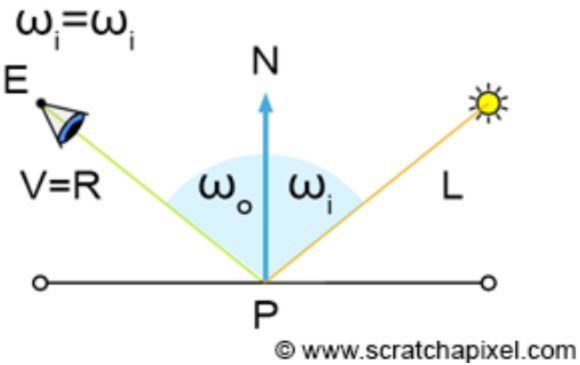
https://help.autodesk.com/cloudhelp/2016/ENU/mental-ray-docs/mr_docs/manual/node102.html

BRDF

- Once amount and direction of incoming light is found, attenuation is performed and then amount of
  computed using **BRDF** `f(L,N`
  - Mathematical model approxima re
    of object material
  - Besides parameters, it depends V
  - Diffuse/Lambertian, specular, Ph
    Cook-Torrance, Ward, Oren-Nayar, etc.
- BRDF contains number of parameters which are varied over surface using **texture**
- General shading; one point light:
  - `shaded_color = f(L, V, N) * light_color * light_intensity * max(0, dotProduct(N, L))`

re-read

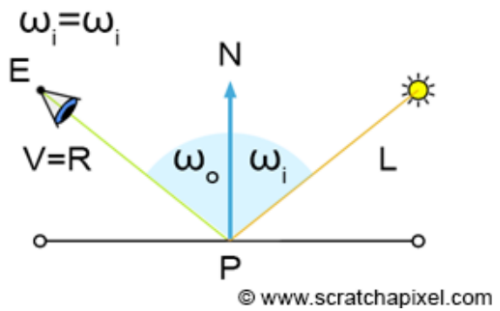


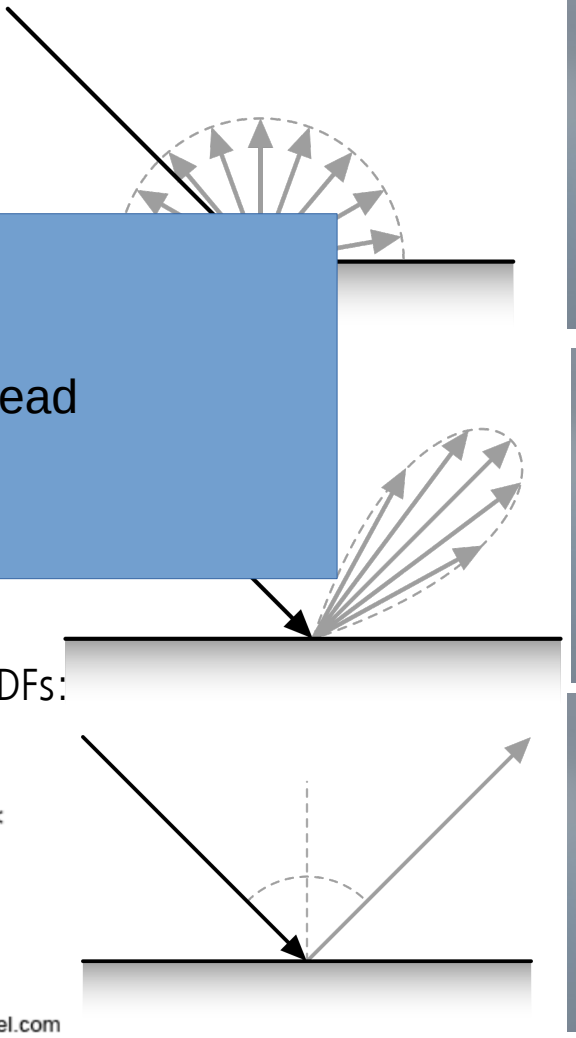


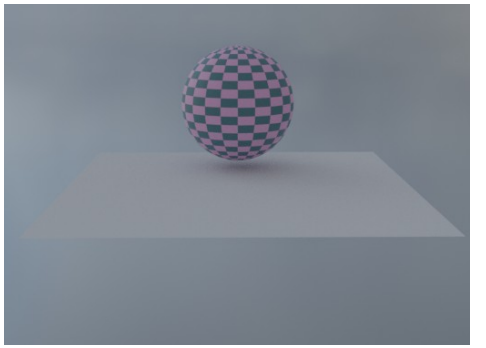
BRDF

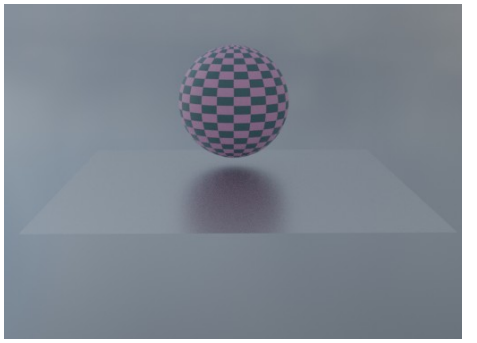
- **Amount reflected light in view direction `v` given indicent direction** `L` - BRDF: `f`
  - L – incoming/incident direction
    - Angle of incidence: ω_i (between
  - V – view direction
    - Line joining eye/camera E and sh
  - R – outgoing/reflected light dire
    - Angle of reflection: ω_o (between N and R)
- Most materials can be described with following BRDFs:
  - Mirror
  - Diffuse/Lamberian
  - Glossy

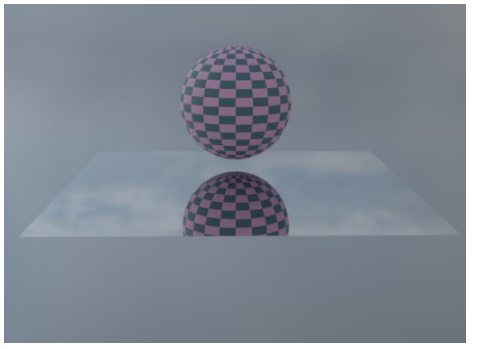
re-read







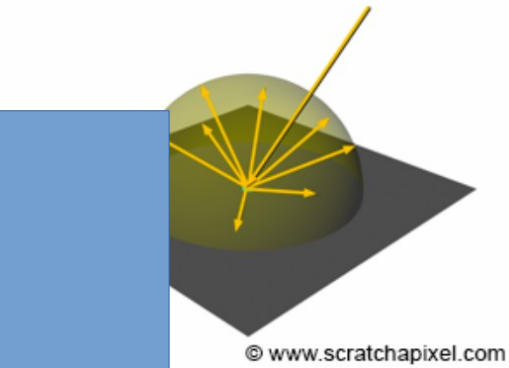


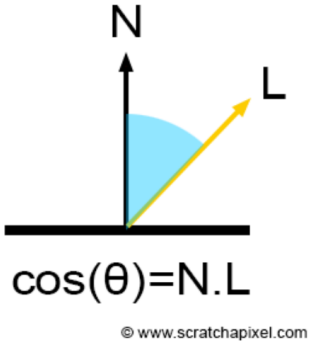


Shading: diffuse/Lambert surface

- Light falling on surface is attenuated based on surface normal and incoming light direction
  - Lambert's cosine law: `dotProduct`
- Diffuse BRDF: `albedo / PI`
  - `albedo RGB in [0,1]`
  - `Albedo = (reflected_lig`
- Diffuse surfaces are reflecting light ir whole hemisphere at shading point P around normal N. Final color:
  - `diffuse_surface = (albedo / PI) * light_color * light_intensity * max(0, dotProduct(N, L))`
- Surface is **view independent**

re-read

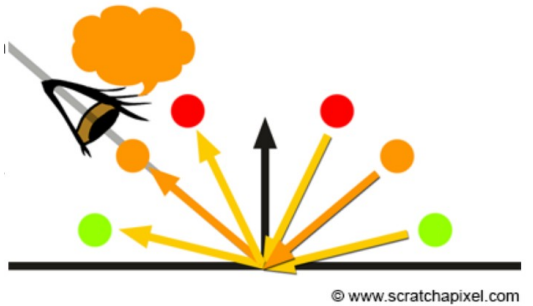


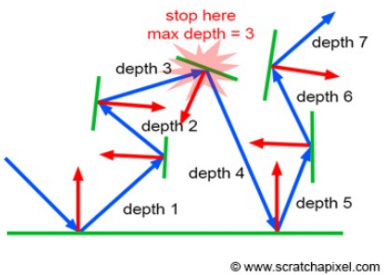


Shading: perfect specular surface

- Light bounces in direction symmetrical to incident direction around normal at shading point → **law of reflection**
  - `incident_angle = re`
- Reflected direction:
  - `R = L - 2 (N * L) *`
- Surface is **view dependent**
- Reflection of mirror surface:
  - Reflection direction is calculated using view direction (primary ray direction) and normal in intersected point
  - Primary ray color (shading result) is equal to color of reflected ray
  - This process is **recursive** until it hits background or non-specular surface → terminate with **ray depth** – trade off between render time and quality
    - Ray-tracing here is solving visibility!

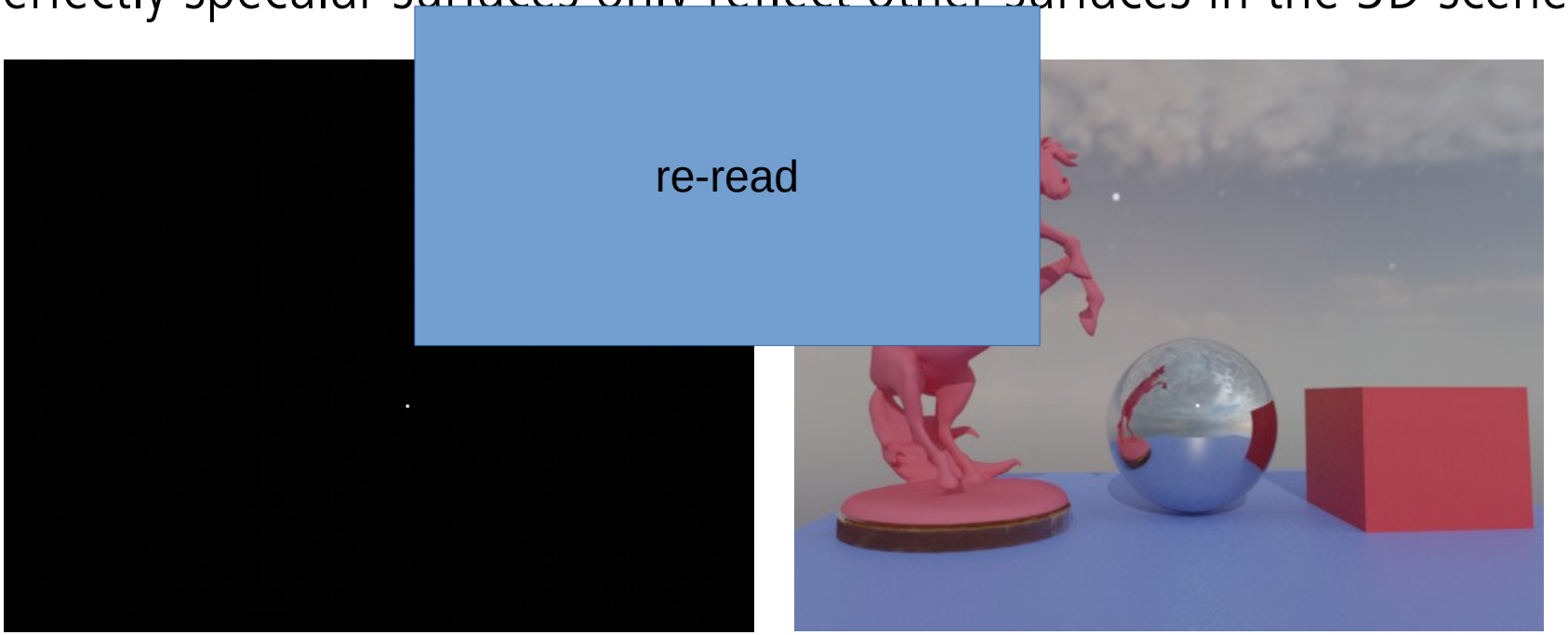
re-read





Shading: perfect specular surface

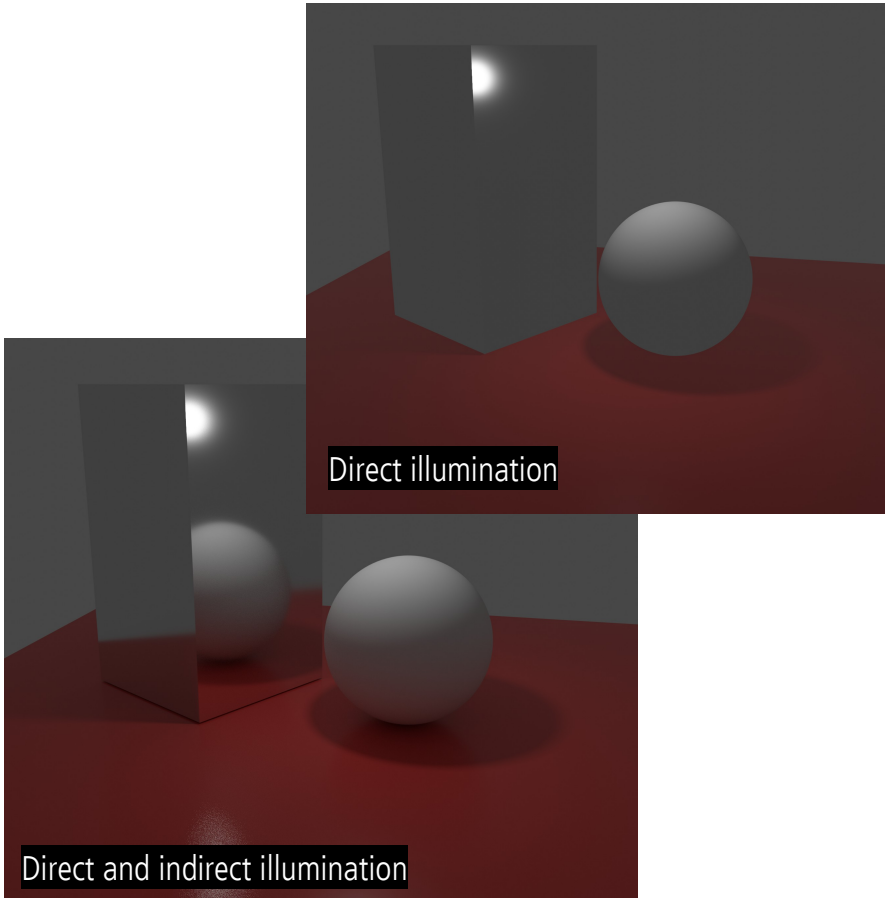
- Perfectly specular surfaces only reflect other surfaces in the 3D scene



Without objects in 3D scene, specular sphere is not visible since it reflects only black color.

Indirect illumination

- Besides light coming directly from light sources, large contribution of light is also indirect
  - Light can travel from light source, reflecting from objects and finally fall on shaded surface → **indirect illumination**
    - **Indirect diffuse**
    - **Indirect specular**
    - **Soft shadows**
- Gathering indirect light on shading point is called **light transport**
  - **Depends on visibility → geometrical problem**



Indirect diffuse – diffuse objects reflect light which illuminates other objects in the scene

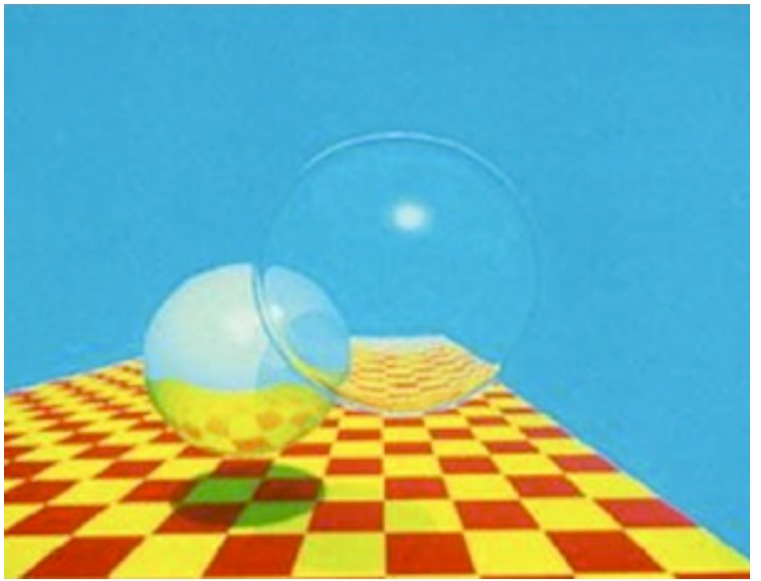
Indirect specular - specular objects reflect light which illuminates other objects in the scene

Shading and light transport

- Appearance of object and can be divided into:
  - **Shading**: how light absorbs and reflects; light-surface interaction
    - How light interacts with matter: what happens to light which reaches the surface and how light leaves the surface
    - Uses scattering model (e.g., BRDF) to compute color
    - Effects: reflection, transparency, specular reflection, diffuse reflection, sub-surface scattering.
  - **Light transport**: how much light object receives
    - How light bounces from surface to surface
    - Which paths light take and how does it depends on material
    - Is light blocked by another surface?
    - Uses scattering model (e.g., BRDF) for computing light reflection
    - Effects: Indirect diffuse, indirect specular, soft shadows
- Distinction between shading and light transport is very thin

Global illumination and light transport

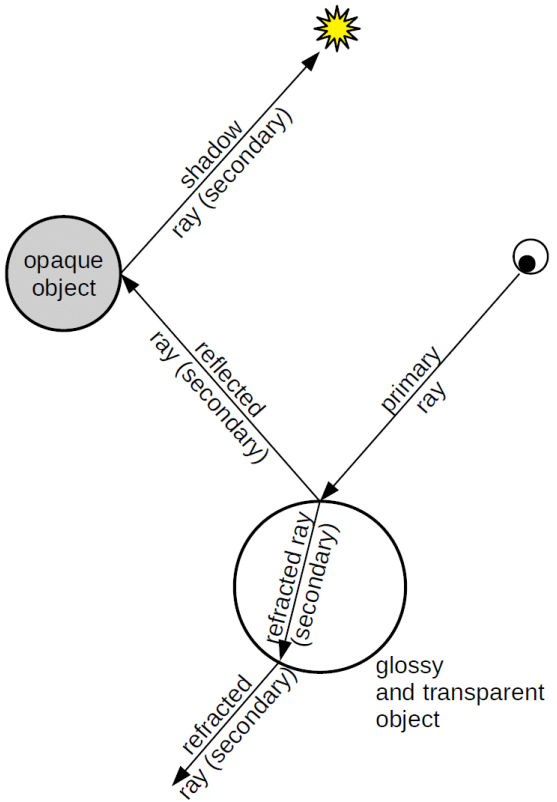
- Rendering equation describes **global illumination**: direct and indirect light
- Various light transport strategies based on ray-tracing for solving full rendering equation, most popular is **path-tracing**
- Classical example of light transport is **Whitted ray-tracing**
  - Not solving full global illumination
  - Utilizing ray-tracing for collecting indirect illumination
  - Advanced light transport strategies build on this method



An Improved Illumination Model for Shaded Display. Turner Whitted, 1980

Whitted ray-tracing

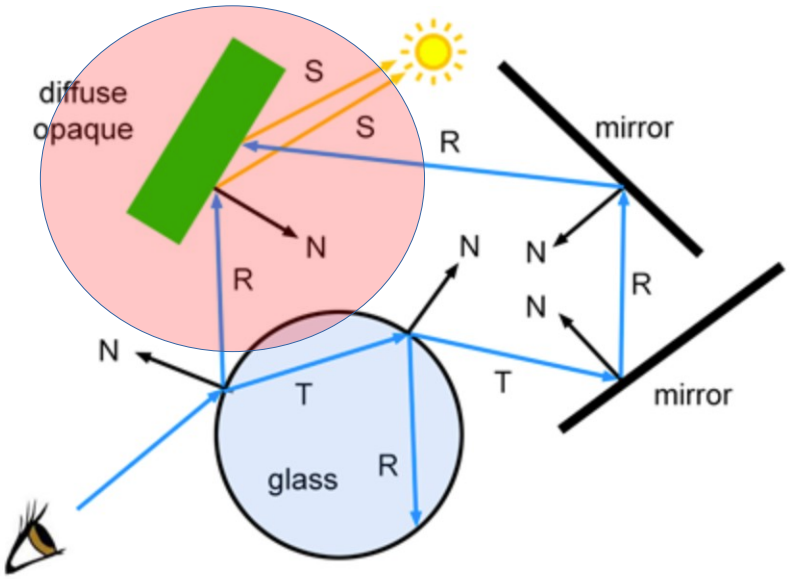
- For each camera ray intersection with object, collect incoming light
- Assumptions for light transport
  - Scene contains only point lights
  - Surfaces are only **specular** (reflection, refraction) or **diffuse**
- Light transport:
  - Laws of reflection and refraction are used to compute direction of light rays intersecting reflective or transparent surfaces → **secondary (shadow) rays**
  - Follow light rays bouncing across the scene and find out color of objects they intersect
- 3 main intersection cases



https://www.researchgate.net/publication/336285790_Heuristic_based_real-time_hybrid_rendering_with_the_use_of_rasterization_and_ray_tracing_method

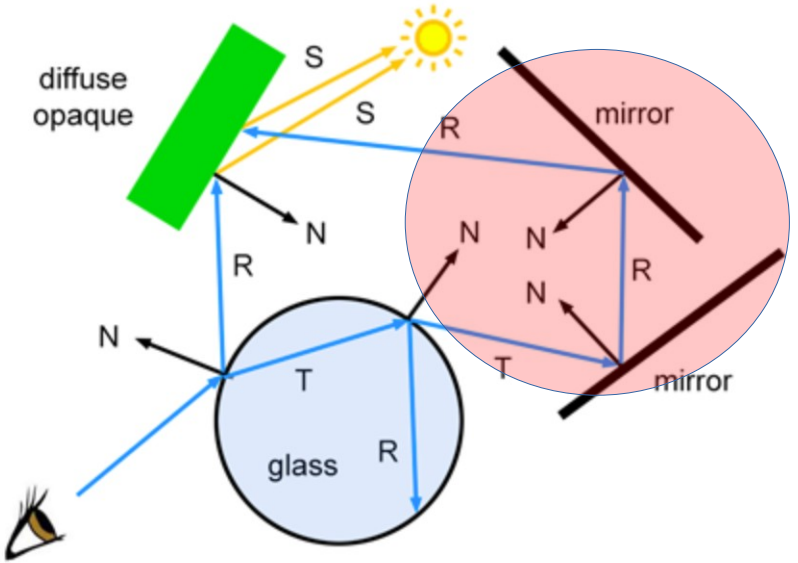
Whitted ray-tracing: case 1

- Surface at intersection is point is **diffuse**:
  - Cast a **shadow ray** in direction of light to find light direction, intensity, color or if it is shadowed
  - Use **scattering model (BRDF) to compute color** of the object: e.g., Lamberitan BRDF.



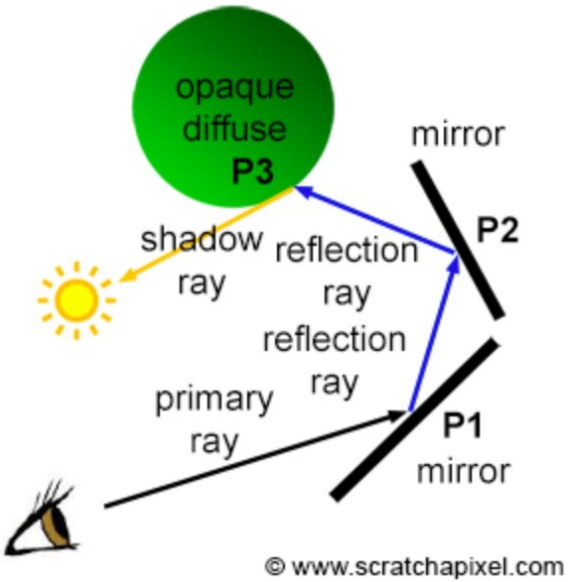
Whitted ray-tracing: case 2

- Surface at intersection is point is **mirror (specular reflection)**:
  - Trace secondary ray – **reflection ray** from intersection point in direction of specular reflection



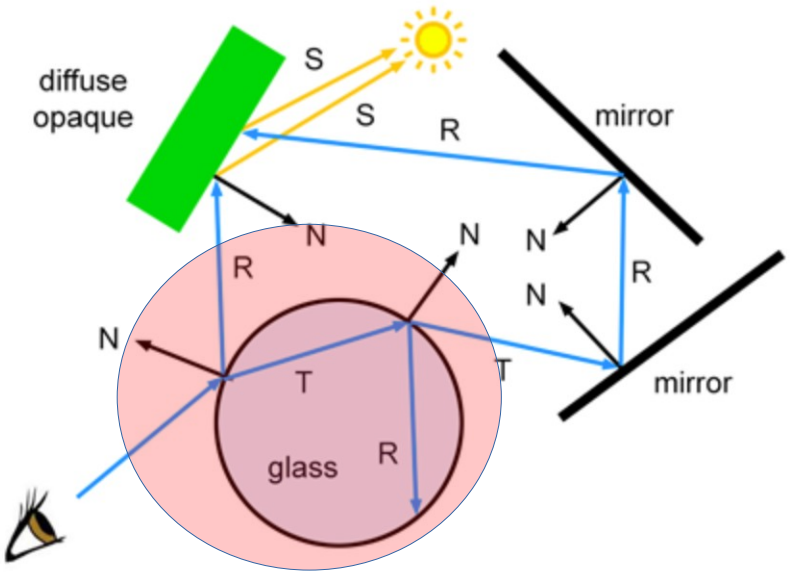
Whitted ray-tracing: reflection ray

- Backward ray-tracing:
  - Primary ray is generated and intersects mirror surface at P1
  - At intersected point P1 reflection ray is generated and intersects mirror surface at P2
  - At intersected point P2 reflection ray is generated and intersects diffuse surface at P3
  - At P3 color of the surface is calculated
- Return color on created path
  - Color at P3 becomes color at P2
  - Color at P2 becomes color at P1
  - Color at P1 is color of primary ray
  - Color of primary ray is color of pixel from which primary ray was generated



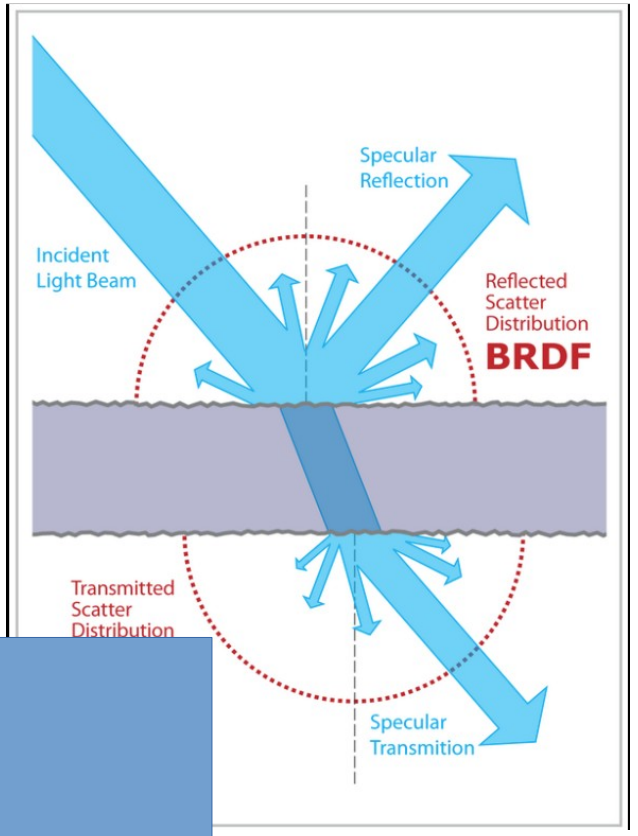
Whitted ray-tracing: case 3

- Surface at intersection is point is **transparent (specular transmission)**:
  - Trace secondary ray – **refraction ray** from intersection point in direction of specular refraction



Shading and BTDF

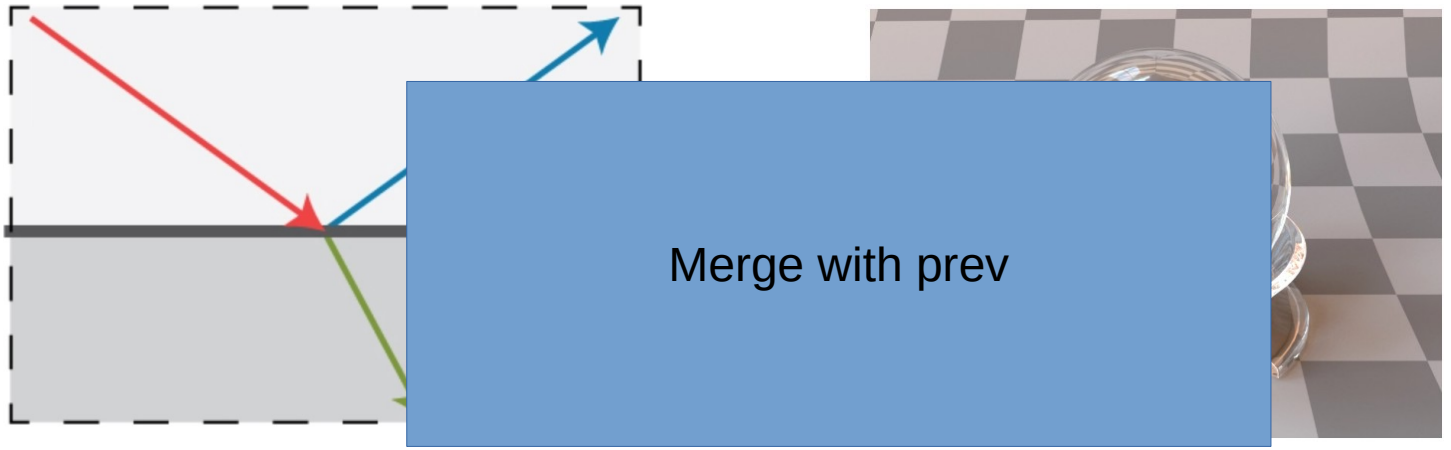
- For transparent objects (e.g., glass), light is reflected and **refracted**
  - Reflection: light changes direction traveling outside of the surface
  - Refraction: light changes direction traveling inside of the surface
  - Transmission: light that enters object on one side and leaves object on other side



Merge with prev

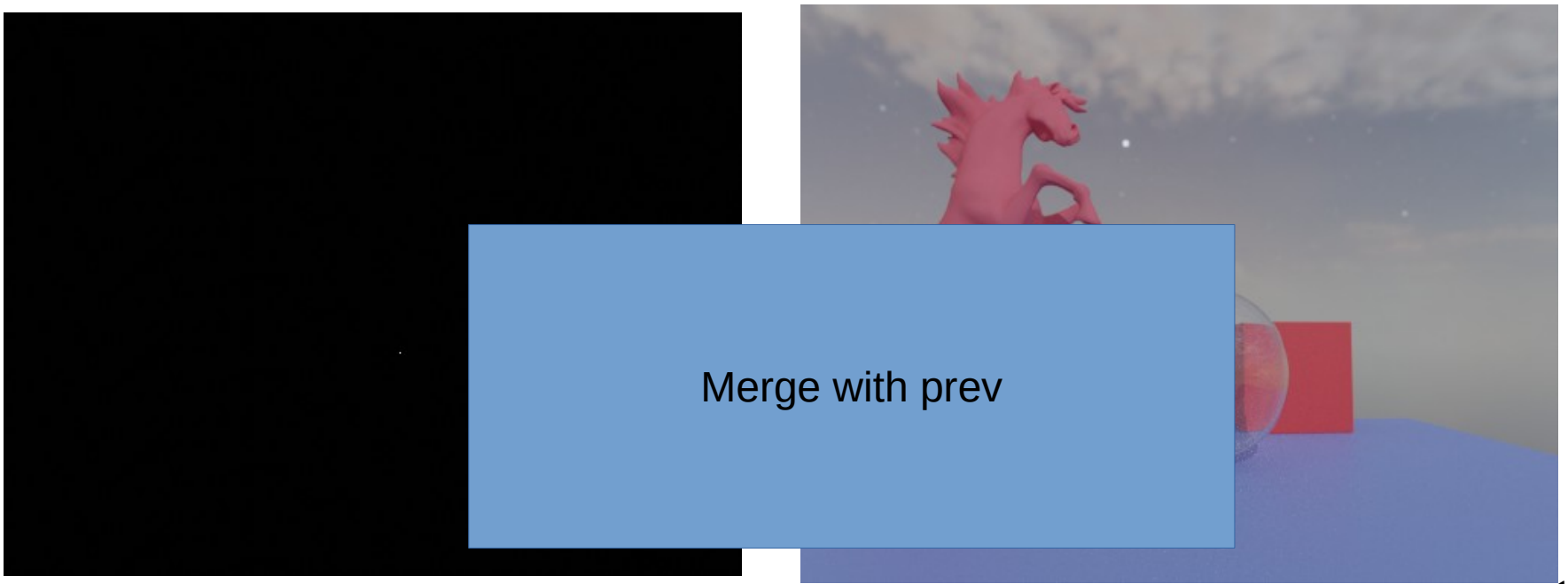
Shading and perfect specular transmission

- Refraction direction is determined by **Snell's law**
- Amount of reflection and refraction is determined by **Fresnel's law**
- Similarly as for specular reflection, specular transmission is recursive tracing since its color depends on objects it reflects



Shading and perfect specular transmission

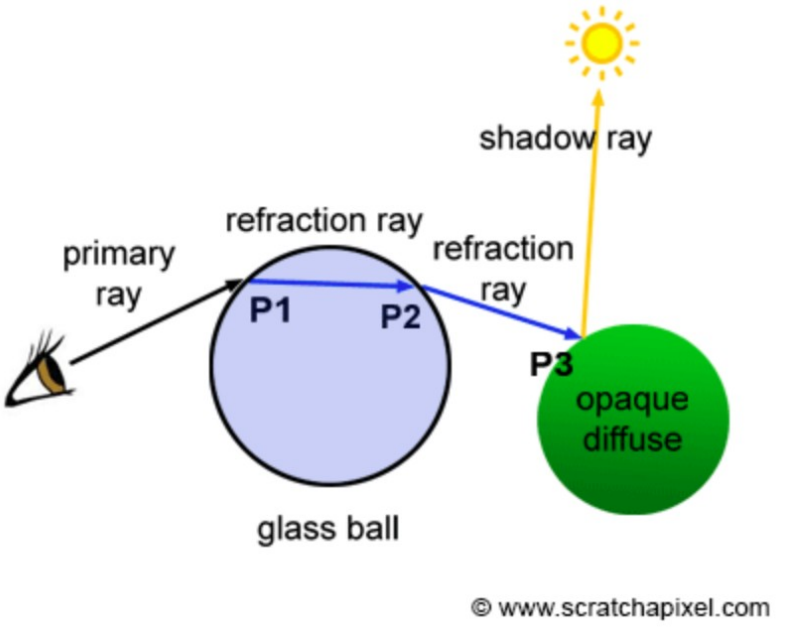
- Perfectly specular surfaces only reflect and transmit other surfaces in the 3D scene



Without objects in 3D scene, specular sphere is not visible since it reflects only black color.

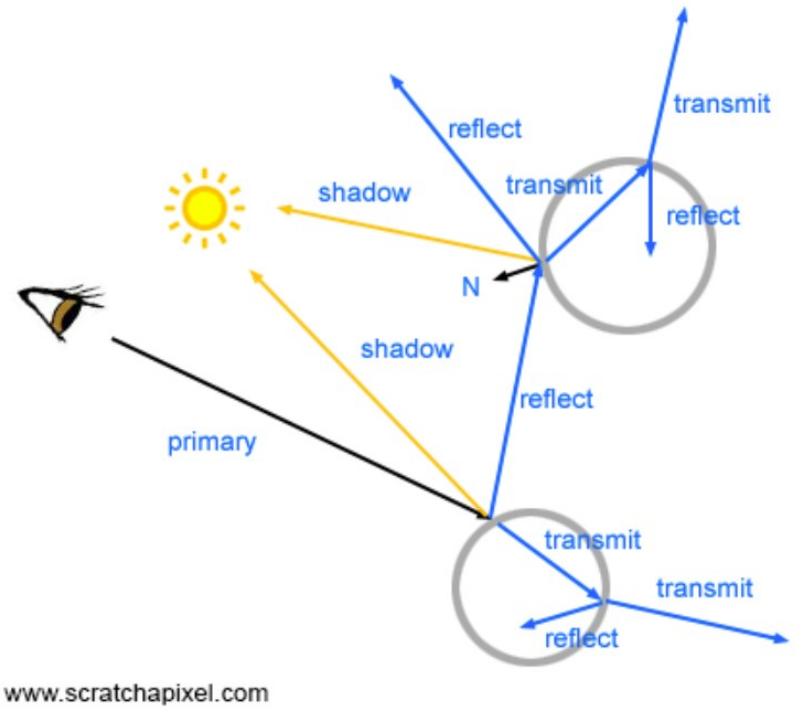
Whitted ray-tracing: refraction ray

- Backward ray-tracing:
  - Primary ray is generated and intersects transparent surface at P1
  - At intersected point P1 refraction ray is generated and intersects
  - At intersected point P2 refraction ray is generated and intersects
  - At P3 color of the surface is calculated
- Return color on created path
  - Color at P3 becomes color at P2
  - Color at P2 becomes color at P1
  - Color at P1 is color of primary ray
  - Color of primary ray is color of pixel from which primary ray was generate



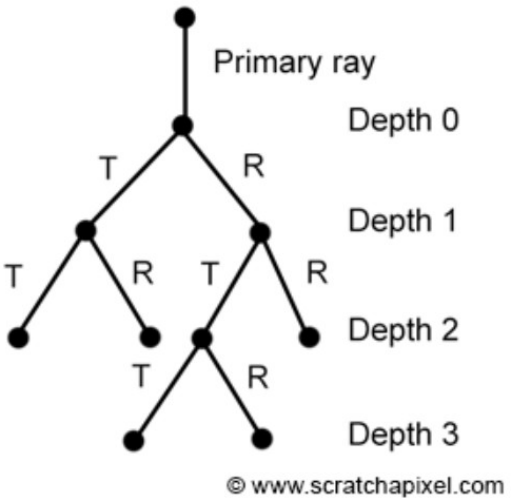
Whitted light-transport properties: recursive nature

- Intersecting reflective surface causes generation of reflection ray
- Intersection of transparent surface causes generation of two new rays: reflected and refracted
- These rays can further intersect reflective or transparent surfaces → **recursion**
  - To evade explosion, max recursion depth is exposed as parameter



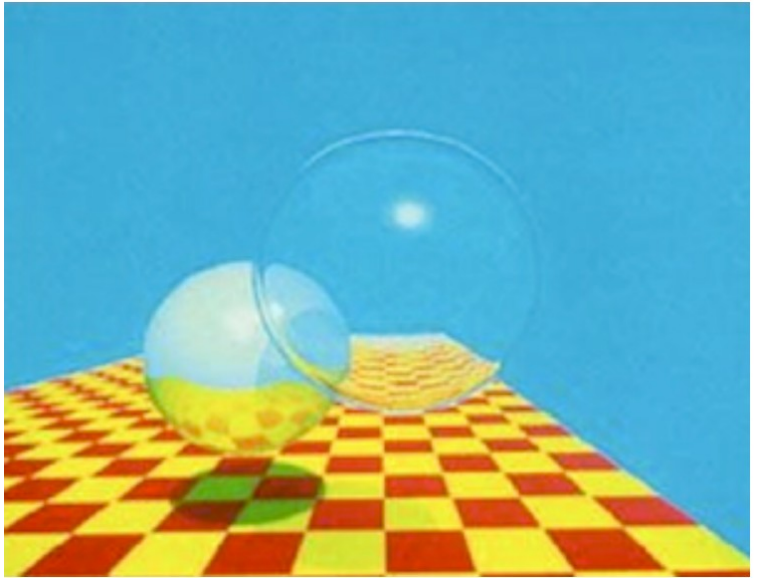
Whitted light-transport properties: tree of rays

- All secondary rays (reflection or refraction) spawned by primary or other secondary rays can be represented tree-like structure
- Each intersection marks new depth/level of the tree and thus one level of recursion



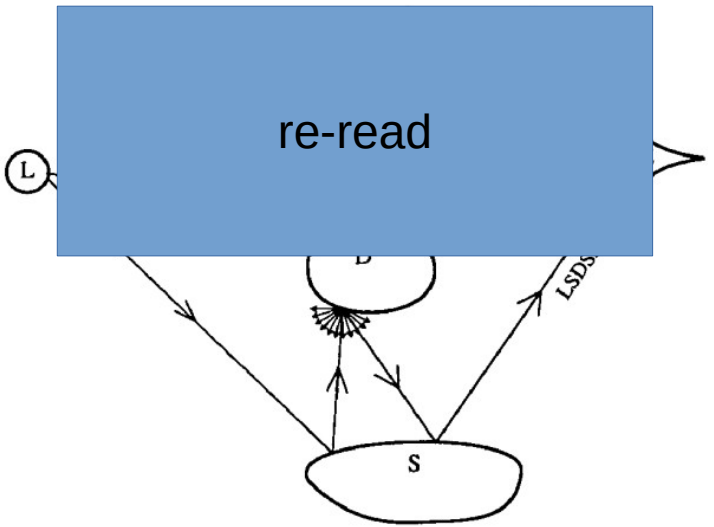
Whitted ray-tracing and light transport

- Whitted ray-tracing - classic light transport algorithm with which rendered image exhibit appearance of:
  - Diffuse objects
  - Mirror-like objects
  - Transparent objects
- Different light transport strategies simulate different effects which is the basis for their categorization



Categorization of light transport strategies

- Light transport strategies vary by light paths (effects) they simulate
- Each intersection of light ray from light source (L) to eye (E) can be:
  - Diffuse (D)
  - Specular (S)
  - Glossy (G)
- Advanced light transport strategies based on ray-tracing:
  - Path-tracing
  - Bidirectional path-tracing
  - Metropolis light transport



https://www.researchgate.net/publication/220721861_Adaptive_Radiosity_Textures_for_Bidirectional_Ray_Tracing

Whitted ray-tracing and beyond

- Whitted ray-tracing is not full solution to global illumination
  - Light reflected from any direction than mirror reflection or refraction is ignored
  - Direct lights are only represented with point lights
- Fully evaluation of global illumination is is proposed by Kajiya as path-t
  - Correct solution which generates global illumination
  - After camera ray intersection is found and during shading evaluation,may rays
    - For diffuse surface, rays over hemisphere at intersection are shot
    - For glossy surface, rays in lobe at intersection are shot
    - Etc.
  - Problem with this approach is explosion of rays, thus Monte Carlo sampling methods are employed for generating paths through environment.
    - Several of such paths are averaged for each pixel
  - General problem with path-tracing is noise and amount of rays that have to be shot

re-read

Ray-tracing based rendering: verdict

- Good:
  - Ray-tracing based rendering is clear and straightforward method to implement*
  - It represent unified framework for computing wha
    (primary ray intersections), with inherent perspecti
    and computing light transport – visibility between
    (secondary ray intersections) required for the shading process.
  - Powerful shading capabilities when it comes to light transport

re-read

* Implementing production ray-tracing based rendering requires a lot of additional features than ones that we have discussed: support for render-time displacement mapping (shape vertices offsetting given height map), motion blur, programmable shading stage (a program which is evaluated for primary ray intersections).

Ray-tracing based rendering: verdict

- Bad:
  - It requires efficient methods for testing ray-shape intersections: sometimes hard to develop (good understanding of mathematics, particularly geometry and linear algebra for geometrical or analytical solutions)
  - Ray-shape intersections test are core element during rendering and they
  - For advanced light transport needed in shading stage, ray-tracing-based must be readly available for intersection testing and material evaluation to compute indirect light or shadows casted by different objects.
  - Additional acceleration datastructures are often desired to accelerate ray-scene intersections. These datastructures require efficient development, cause additional precomputations (with tremendous acceleration times, though) and require a lot of memory – even more for animated scenes.

re-read

* As we will discuss, this is not needed for rasterizer since only visible geometry is taken in account – which on the other hand disables advanced light transport (again, visibility calculation between objects in the scene) and thus shading.

More into topic

- Generally about ray-tracing:
  - https://www.scratchapixel.com
  - https://raytracing.github.io/books/RayTracingInOneWeekend.html#overview
- Acceleration datastructures:
  - https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-acceleration-structure/bounding-volume-hierarchy-BVH-part2.html
  - https://www.pbr-book.org/3ed-2018/Primitives_and_Intersection_Acceleration/Bounding_Volume_Hierarchies
  - https://www.realtimerendering.com/