Lecture 12: Ray-tracing based rendering

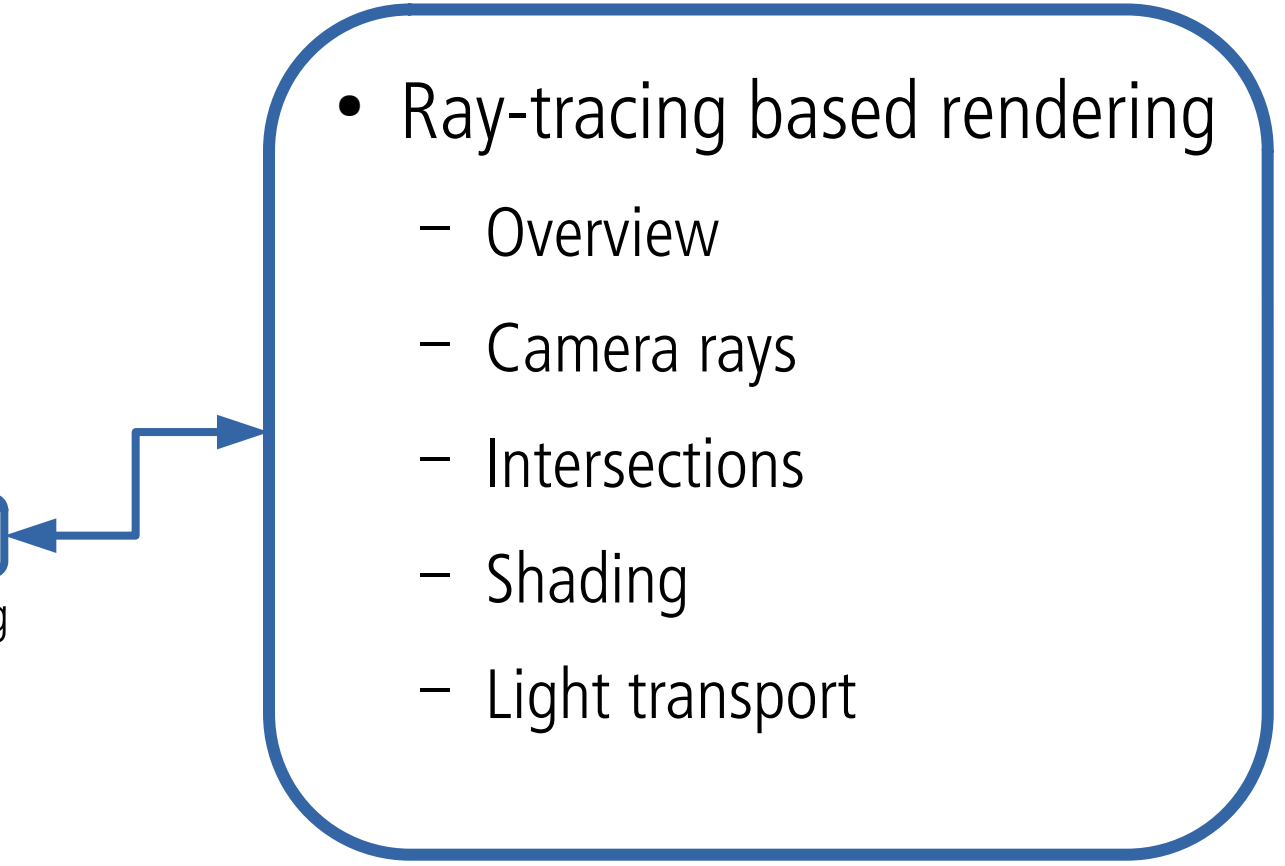
DHBW, Computer Graphics

Lovro Bosnar

8.3.2023.

Syllabus

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

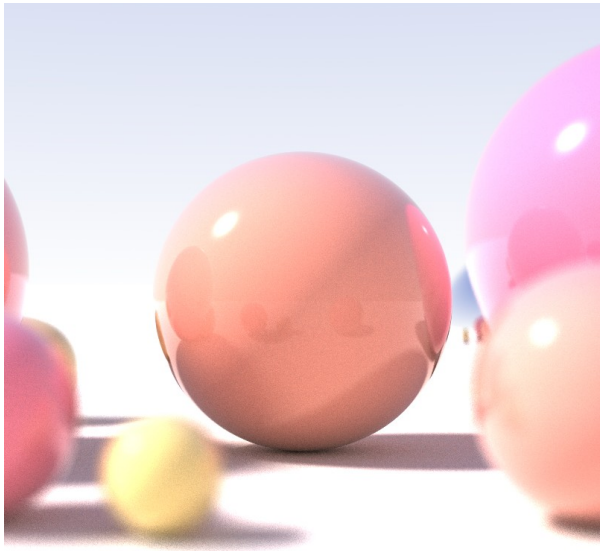


Ray-tracing-based rendering:
https://www.realtimerendering.com/Real-Time_Rendering_4th-Real-Time_Ray_Tracing.pdf

Raytracing-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 light, view, shape and material information

Ray-tracing-based rendering

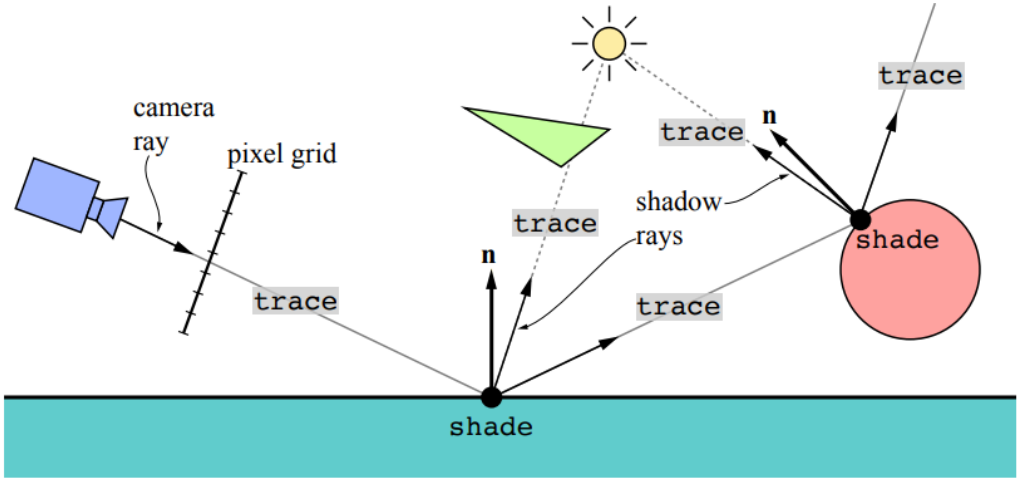
- **Visibility problem**: two points are visible to each other if line segment that joins them does not intersect any obstacle.
- Ray-tracing fundamentally **solves visibility problem using ray-casting**
  - **Finding objects visible from camera**: generate ray from camera and test intersections with 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 obtain what is visible from camera
- **Shading**: calculating the color and intensity of intersected points
  - Light-matter calculation
  - Light transport for incoming light

Ray-tracing-based rendering: trace & shade

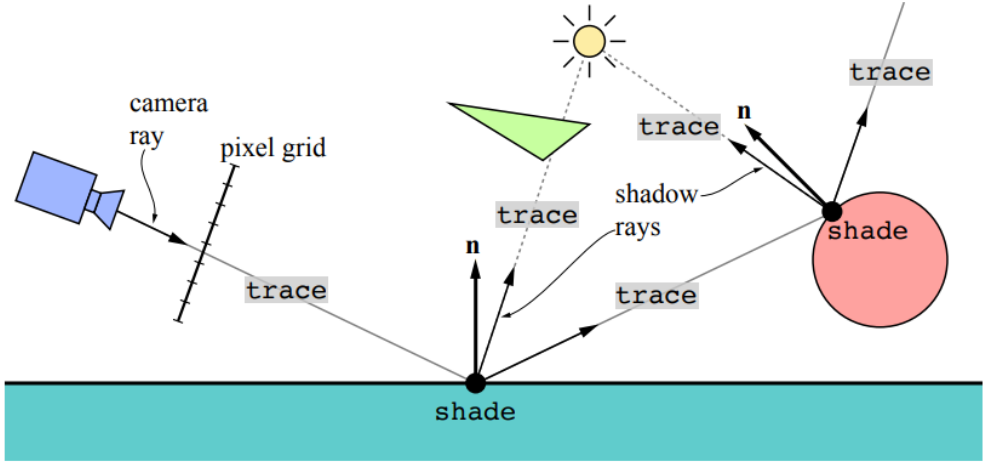
- Raytracing can be described with two functions **`trace()`** and **`shade()`**
- **`trace()`** is geometrical part of algorithm responsible for finding closest intersection between ray and the objects in 3D scene
- **`Shade()`** returns color of the ray intersecting object found by **`trace()`**



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

`trace()` for camera rays

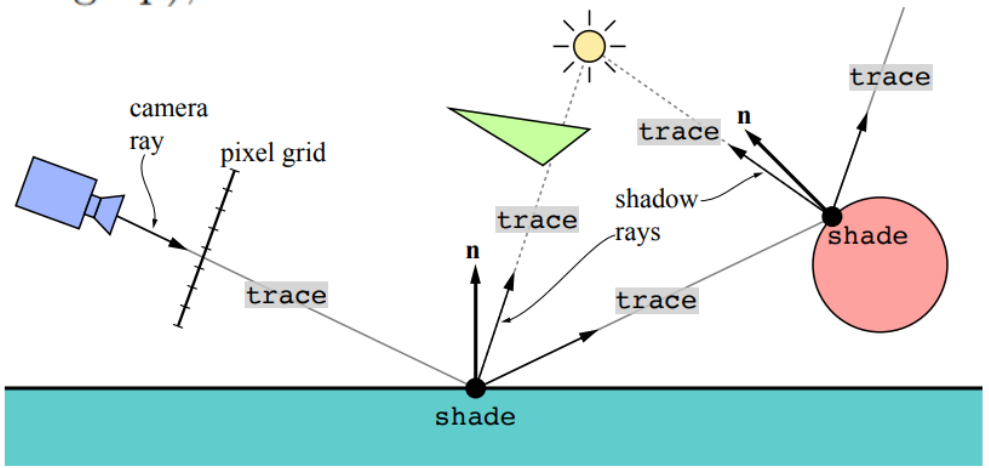
- Rendering starts by generating camera rays for each pixel in the image
- `trace()` function is used on generated camera rays
  - Find closest intersection of camera ray with 3D scene objects
  - Naive `trace()` function: loop through all n objects in the scene and returns closest intersection
    - `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
```

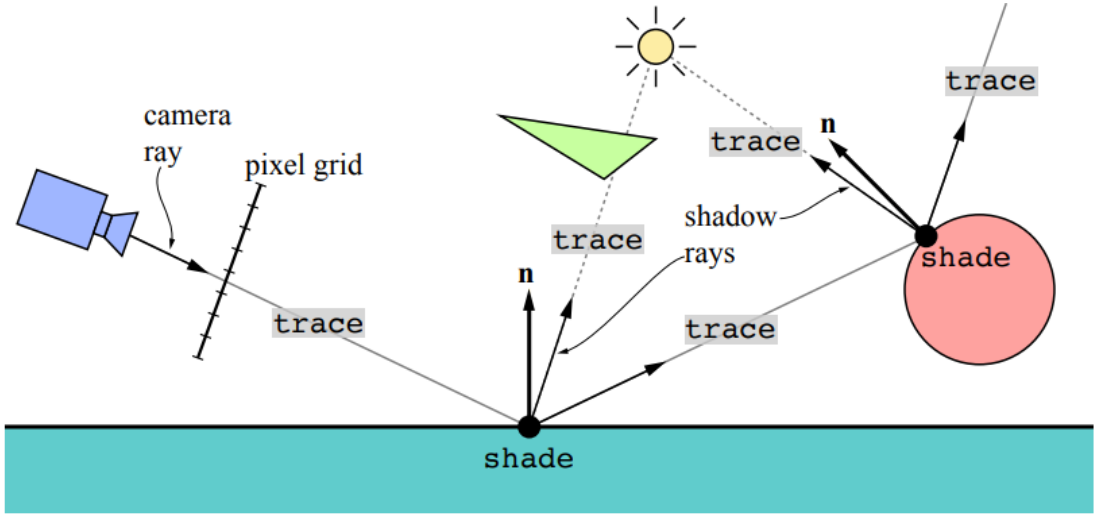
```
function TRACE(ray)
    pt = find closest intersection;
    return SHADE(pt);
end function
```



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

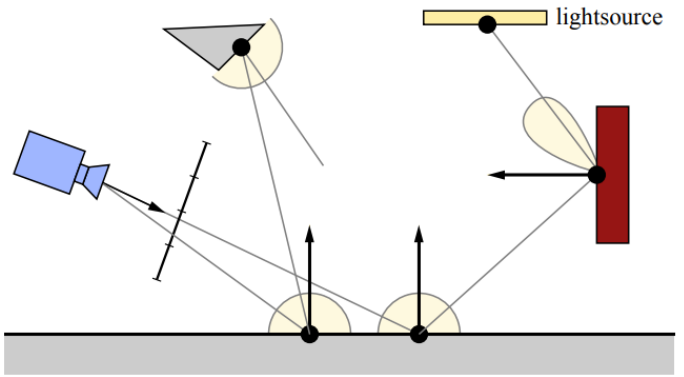
`shade()` for intersections

- `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

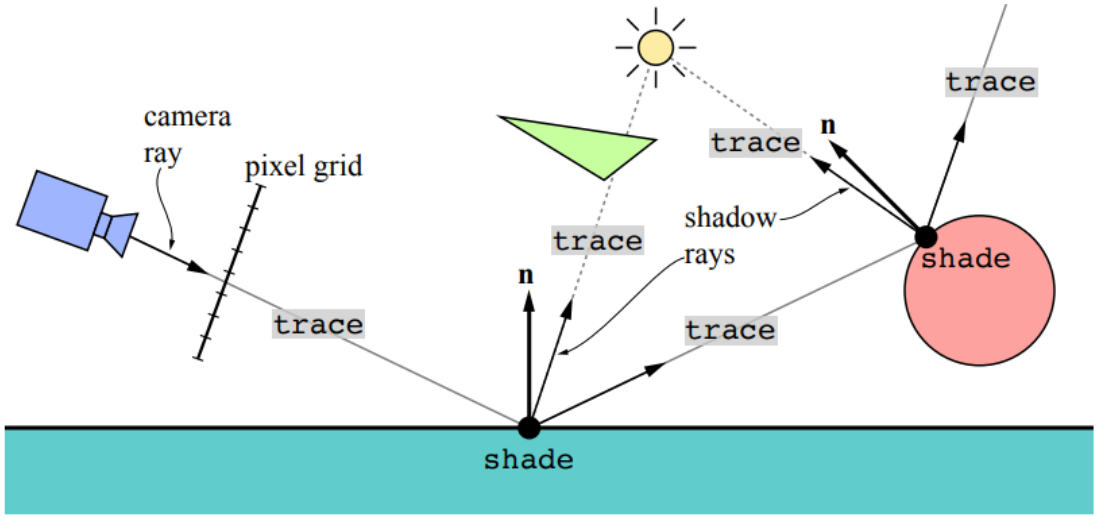
- Light transport: obtaining incident light on shading point
- `trace()` can calculate visibility between shaded point and light source – shadow ray
- `trace()` can compute reflection or refraction ray



`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
```

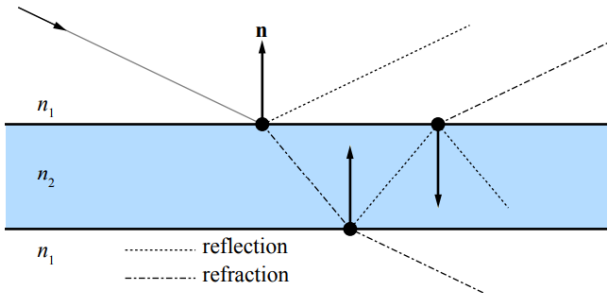
```
function TRACE(ray)
    pt = find closest intersection;
    return SHADE(pt);
end function
```



- 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()`

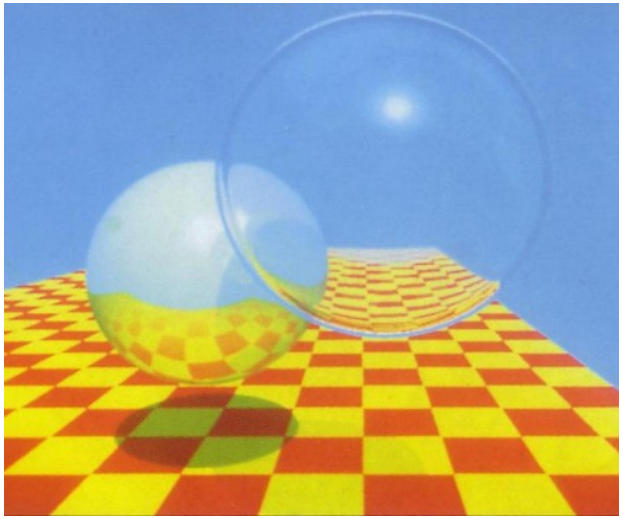
- `shade()` function is implemented by user as a **shader** program
  - Material (scattering functions and textures) is used to light-surface interaction, that is, color
- Traversal and intersection testing which takes place in `trace()` function is often implemented on CPU
  - GPU rendering uses compute 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**
  - Whitted ray-tracing consists of perfectly sharp (specular) reflections and refractions
- Whitted ray-tracing is foundation of many other rendering variants such as path-tracing which are solving rendering equation and global illumination.



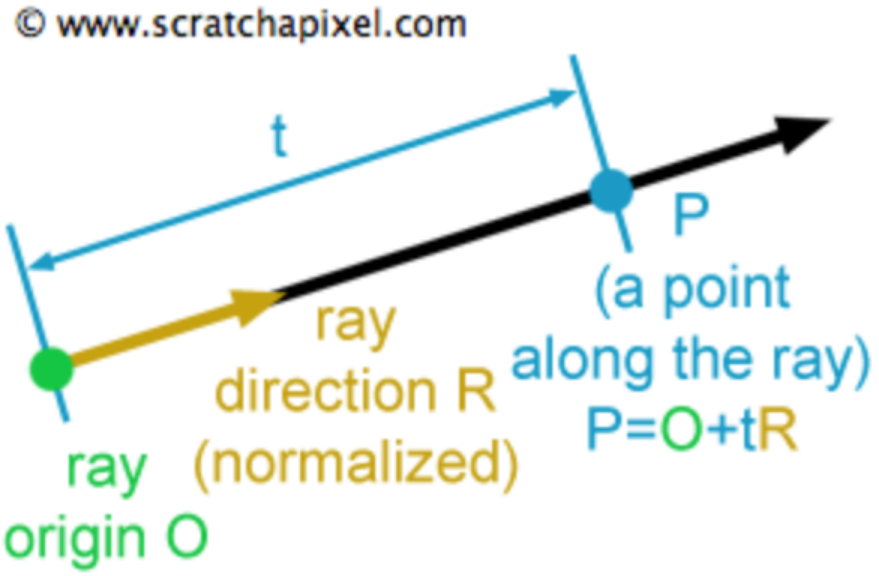
Ray-tracing based rendering

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:

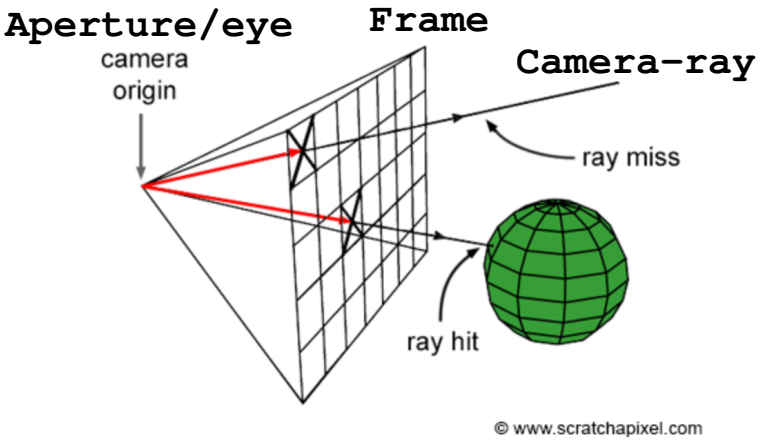
```
P(t) = origin + t * direction;
```

  - `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`



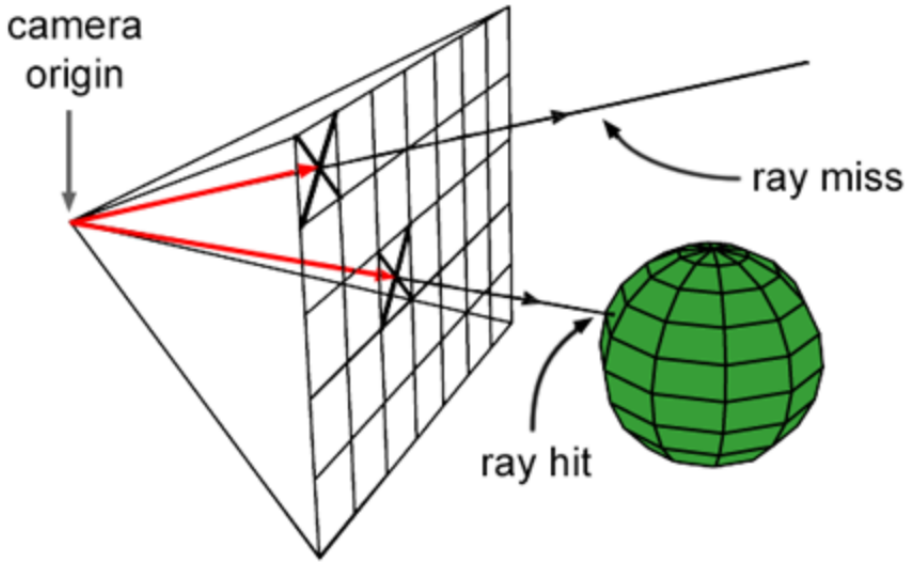
Generating camera rays

- 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**



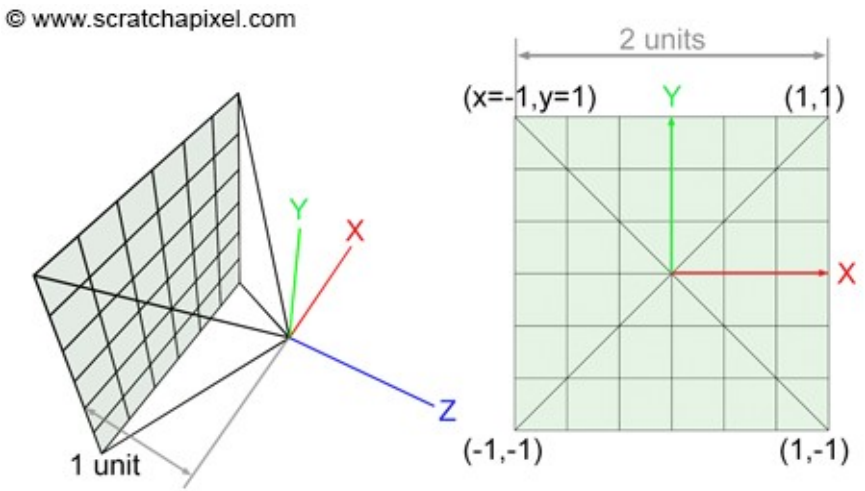
Generating camera rays

```
Vector3f ImageBuffer[imageWidth, imageHeight];
For (int j = 0; j < imageHeight; ++j)
{
    For (int i = 0; i < imageWidth; ++i)
    {
        For  (int k = 0; k < nObjectsInScene; ++k)
        {
            Ray ray = buildCameraRay(i, j);
            if (intersect(ray, object[k])
            {
                // Object hit. Compute shading…
                ImageBuffer[j * imageWidth + i] = shadingResul
            }
            if (intersect(ray, object[k])
            {
                // 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`)
- **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**



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]**

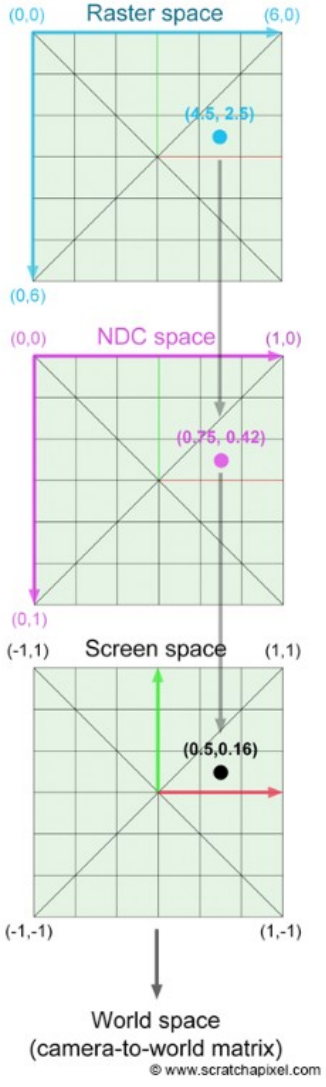
$$PixelNDC_x = \frac{(Pixel_x + 0.5)}{ImageWidth},$$

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

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

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

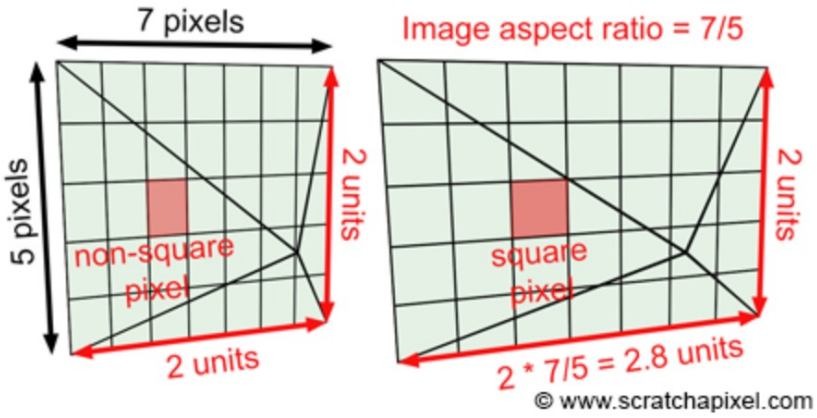
$$PixelScreen_y = 1 - 2 * PixelNDC_y.$$



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

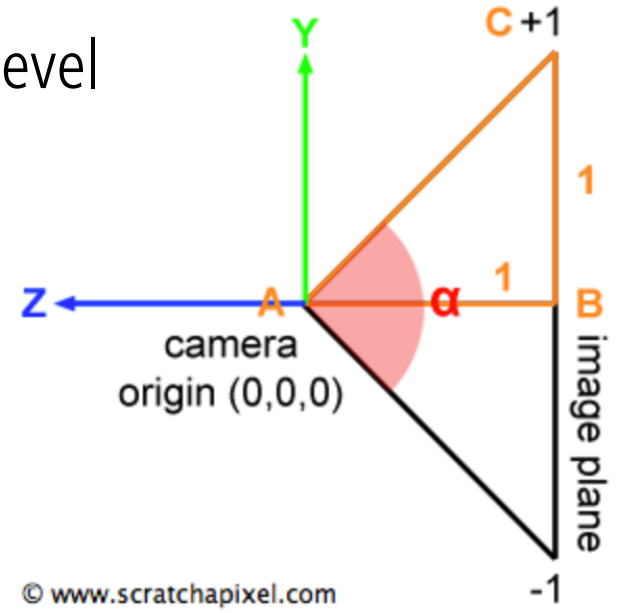
$$ImageAspectRatio = \frac{ImageWidth}{ImageHeight},$$
$$PixelCamera_x = (\quad PixelScreen_x \quad) * ImageAspectRatio,$$
$$PixelCamera_y = (\quad PixelScreen_y \quad).$$



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

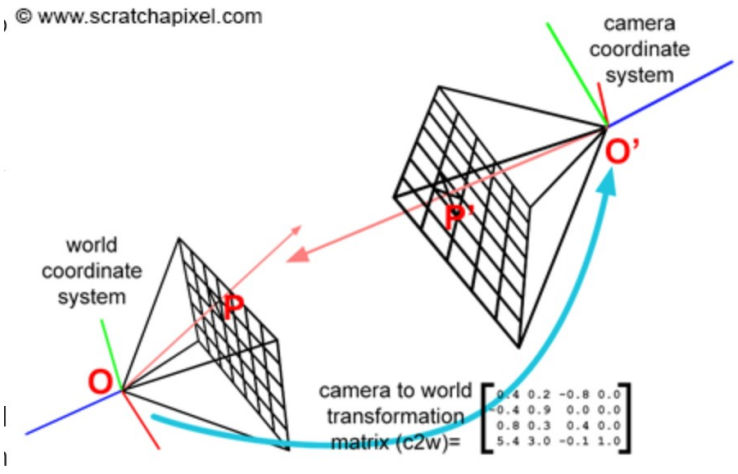
$$P_{cameraSpace} = (PixelCamera_x, PixelCamera_y, -1)$$

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 – cameraOrigin);`
- `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.

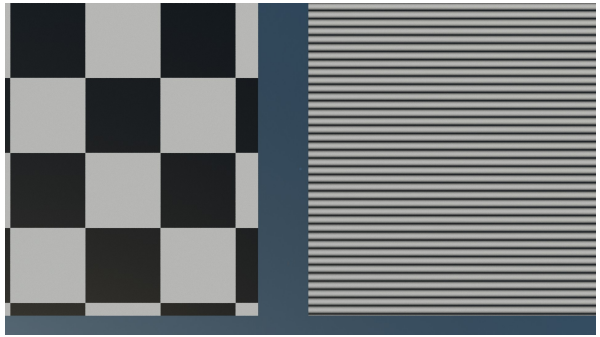
Generating camera rays: multiple rays

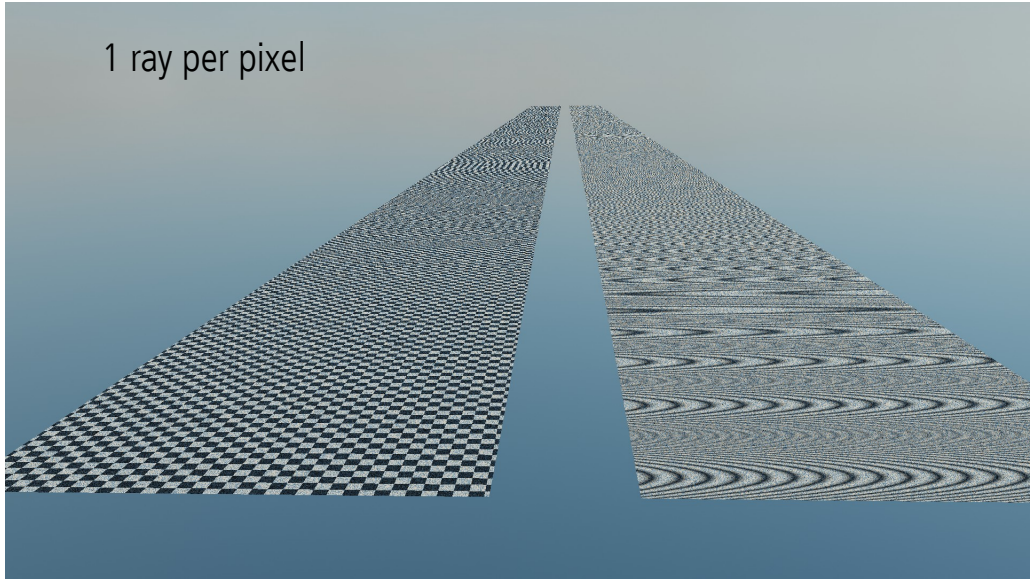
- It is possible, and in practice always desired, to generate multiple rays for each pixel.
- It is important to note that large portion of the scene is actually represented by only one pixel.
- Since pixel can represent only one color, it is important to use multiple rays to obtain the color wh scene covered by the 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)
- Using multiple rays per pixels, reduces:
  - Noise

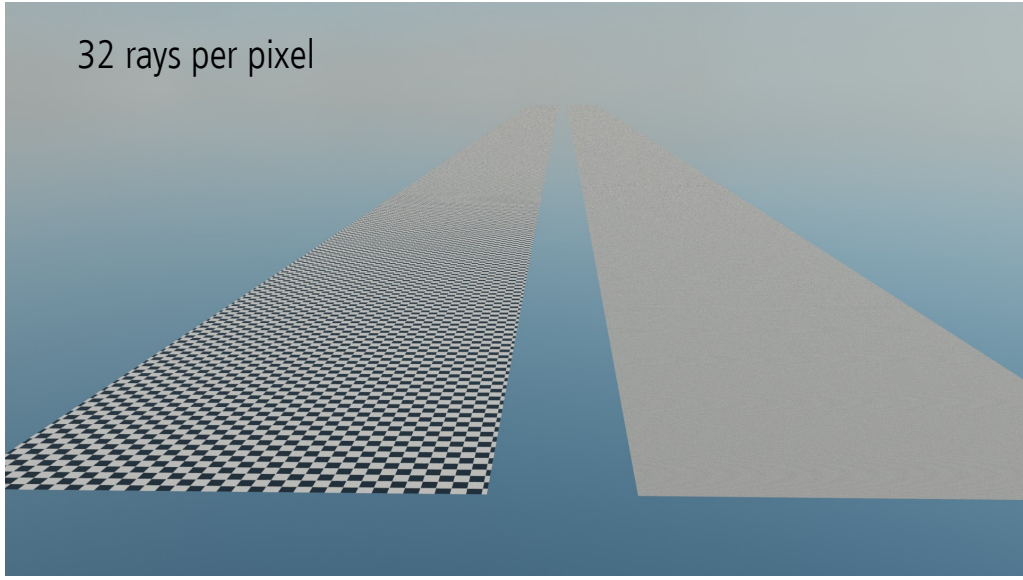
CONTINUE

Generating camera rays: multiple rays

- Using multiple rays per pixels, reduces:
  - Aliasing







Ray-tracing: image centric method

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

Camera rays: testing for intersections

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

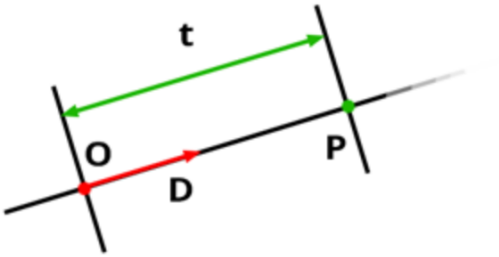
- Generated camera rays are **traced** into the 3D scene
  - Loop over all 3D objects in scene and test them for intersection with camera ray → visibility test!
- In this step, we are interested in a shape of 3D object.
- Once intersection is determined material will be used during the shading step.
- Usefulness of decoupling material and shape of 3D object (if one decides on such rendering architecture decision).

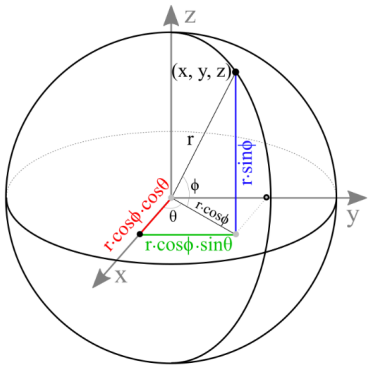
Testing object intersections

- Objects in 3D scene, come in **various shape (geometry) representations**
  - Parametric representations
    - Spheres, disks, planes, etc.
    - Surfaces and curves (e.g., Bezier curves, NURBS, etc.)
  - Polygonal mesh (triangulated, quad, etc.)
  - Implicit surfaces: spheres, SDFs
  - Subivision surfaces
  - Voxels
  - Etc.

Ray-sphere intersection

- Ray-sphere intersection is simplest ray-geometry intersection
  - Very often, ray-tracing demo scenes contain spheres
- 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 point on a sphere:
    - $P^2 - R^2 = 0$ → implicit function which defines implicit shape: sphere





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

Ray-sphere intersection

- 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: $f(x) = ax^2 + bx + c$:
  - $a = D^2$, $b = 2OD$, $c = O^2 - R^2$, $x = t$. Solution:

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

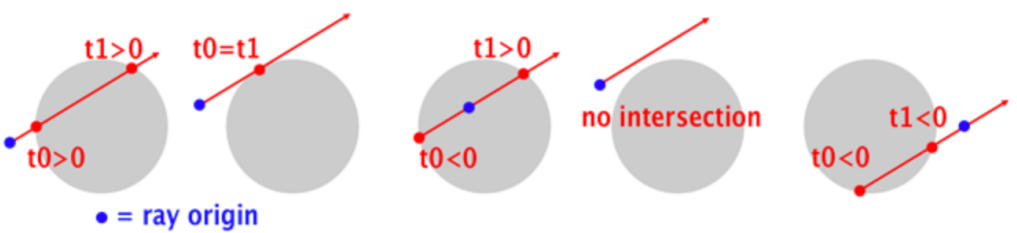
  - $\Delta > 0$: ray intersects sphere in two points (t_0 and t_1):

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

  - $\Delta = 0$: 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`:
  - $a = D^2 = 1$ (ray direction `D` is normalized)
  - $b = 2D(O - C)$
  - $c = |O - C|^2 - R^2$

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

$$\Delta = b^2 - 4ac$$

Ray-sphere intersection: intersection point

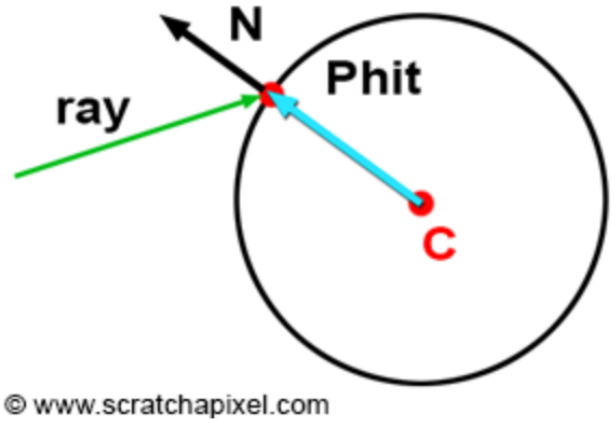
- Now we know t_0 – closest intersection of ray with a sphere
- To find intersection point, we use parametric ray equation:
  - `P(t) = O + t * D`
- Finally,
  - `vector3 Phit = O + D * t`

Ray-sphere intersection: intersection context

- Next to intersection point `Phit`, renderer often computes additional information regarding intersection – **intersection context:**
  - Normal in intersection point
  - Texture coordinate in intersection point
  - Etc.
- Intersection context contains information which is used further in rendering process, e.g., shading

Ray-sphere intersection: normal

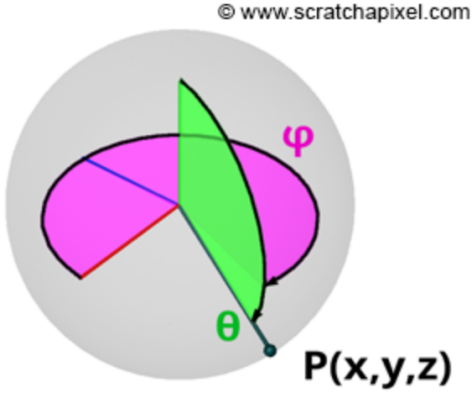
- Normal calculation in intersection point depends on shape representation
- For implicit sphere:
  - `vector3 N = normalize(Phit - C)`



Ray-sphere intersection: texture coordinates

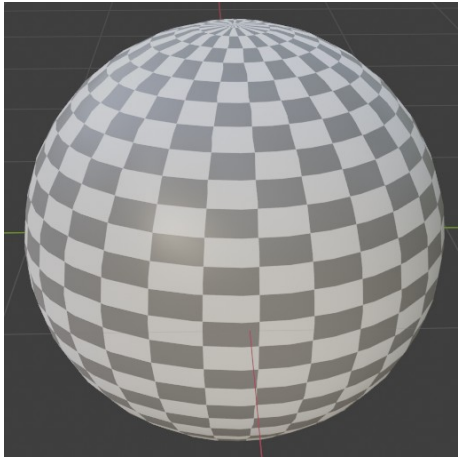
- Calculation of texture coordinates depends on shape representation
  - For parametric and implicit surfaces it can be calculated on the fly
  - For polygonal (e.g., triangle) mesh, texture coordinates are stored per vertex. Intersection point is used to interpolate texture coordinates per triangle face.
- Sphere can be written in parametric form:

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



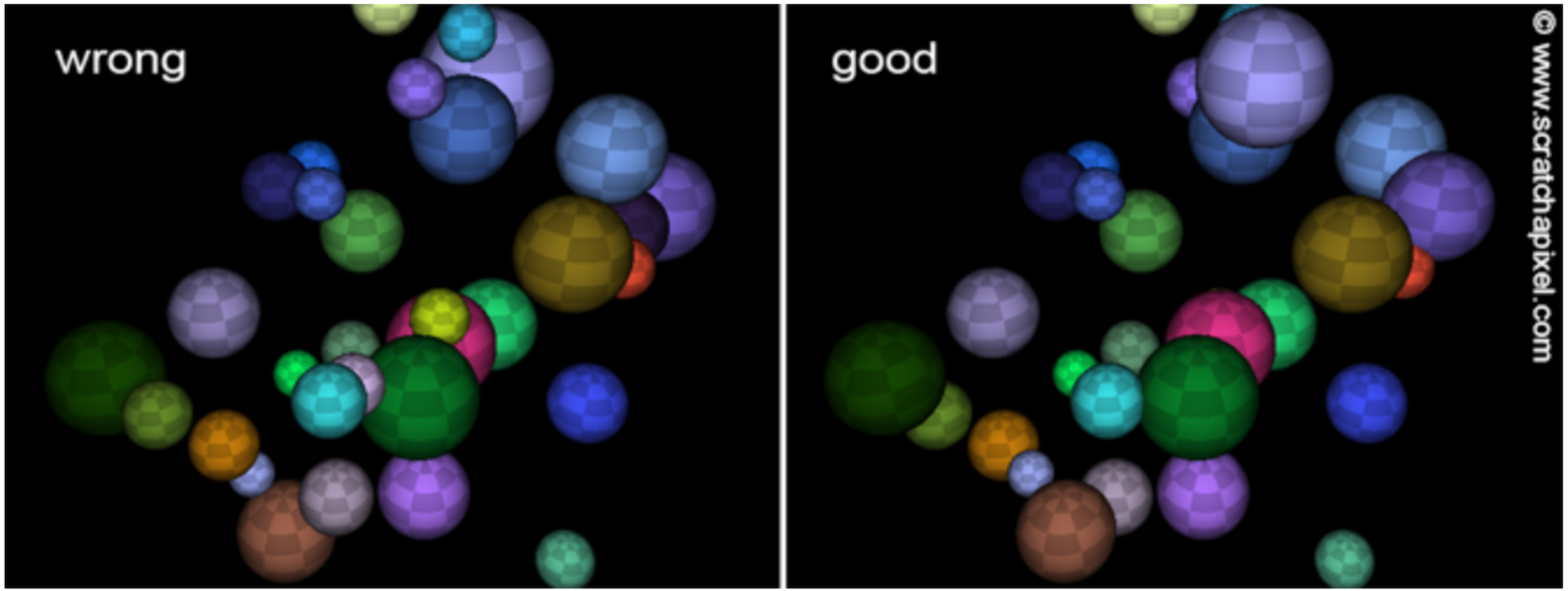
- Texture coordinates for sphere are simply spherical coordinate

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



Ray-sphere intersection: intersection order

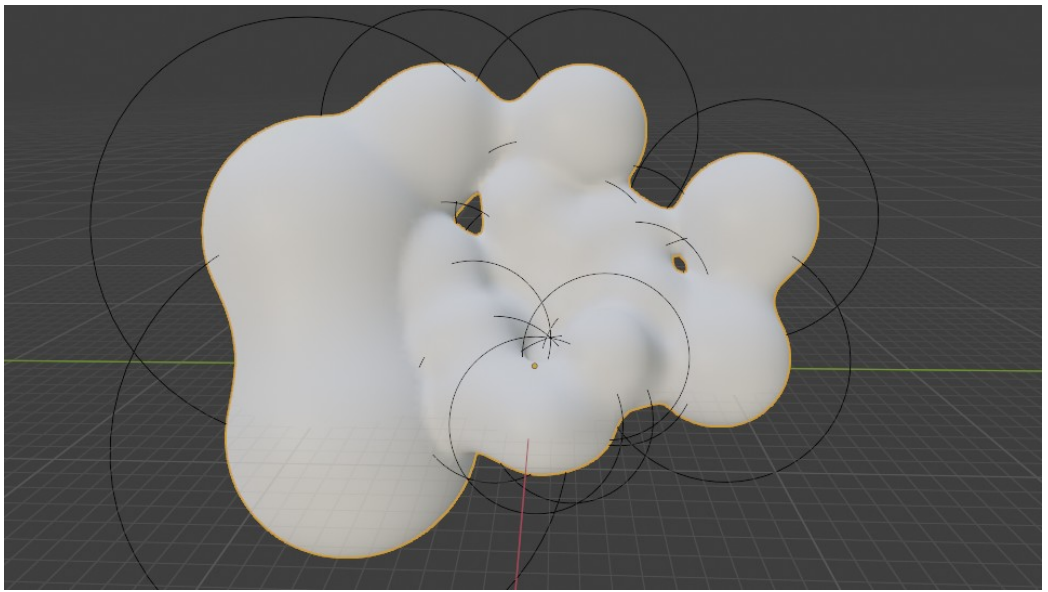
- If scene contains multiple objects then certain depth sorting is needed
- The solution is to keep track of closest intersection distance → closest t

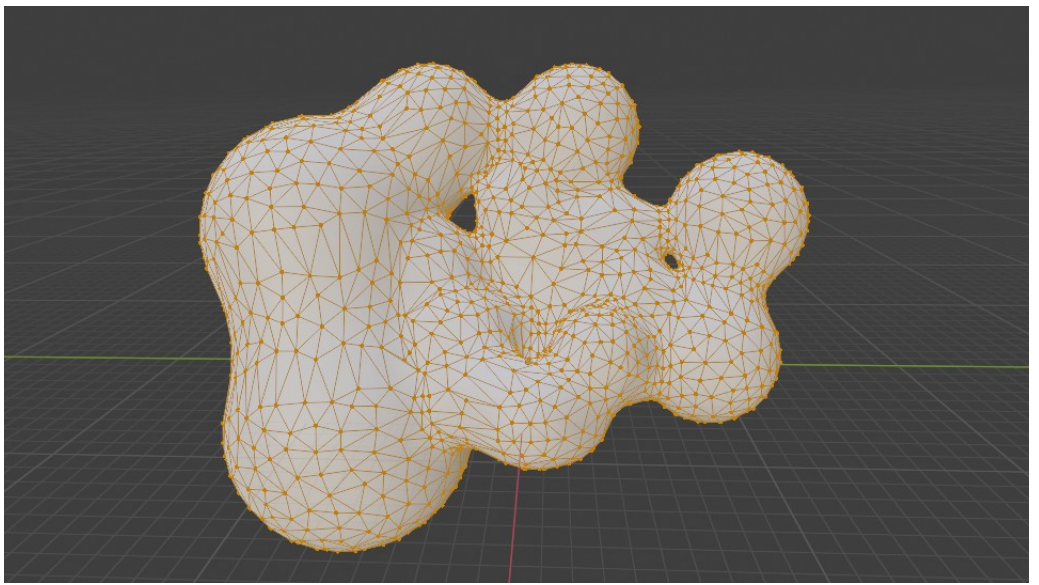


Intersecting other shapes

- Ray-shape intersection must be defined for each representation separately*.
  - e.g., different intersection test is performed for triangle meshes, quad meshes, parametric surfaces, etc.
- Alternative solution: convert each shape representation to same internal representation which is used for rendering.
  - This approach is taken by almost all professional rendering software
  - Internal representation is in almost all cases **triangulated mesh**
  - Process of converting different shape representations to triangulated mesh called **tessellation**

Implicit spheres (metaballs) converted into triangle mesh





* https://www.realtimerendering.com/intersections.html

Reading: 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. G
  - 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
  - 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).

Camera rays: testing for intersections

```
…
for  (int k = 0; k < nObjectsInScene; ++k)
{
    for (int n = 0; m < objects[k].nTriangles; ++n)
    {
        Ray ray = buildCameraRay(i, j);
        if (intersect(ray, object[k].triangles[n])
        {
            // Object hit. Compute shading…
            framebuffer[j * imageWidth + i] = shade();
        }
        if (intersect(ray, object[k].triangles[n])
        {
            // Background hit. Compute background color...
            framebuffer[j * imageWidth + i] = backgroundColor;
        }
    }
}
...
```

- Generated camera rays are **traced** into the 3D scene
- Loop over all 3D object triangles in scene and test them for intersection with camera ray → visibility test!

Intersecting triangle primitive

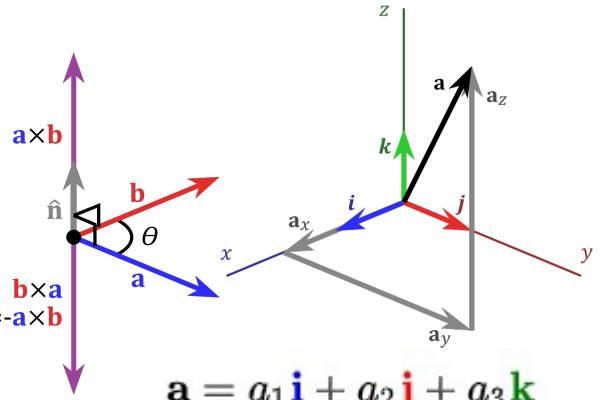
- Basic ray-triangle intersection is straight forward, complexity comes due to multitude of different cases which must be accounted for
  - Thus, there are several algorithms that have been developed and are being developed
- Main questions:
  - Does ray intersect a plane defined by a triangle?
  - Does ray intersect point inside the triangle?

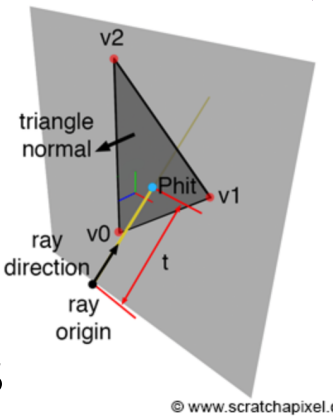
Intersecting triangle primitive: tools

- Triangle vertices are lying on a plane
  - Triangle is coplanar and defines a plane
- Using triangle vertices, normal can be computed
  - Plane defined with triangle 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!**





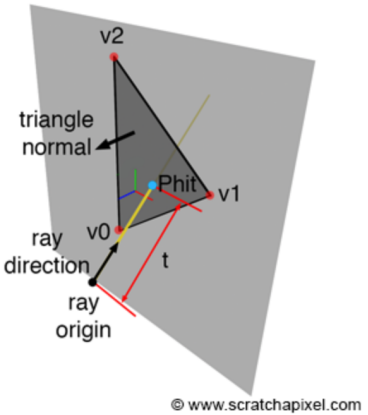
$$\mathbf{a} = a_1\mathbf{i} + a_2\mathbf{j} + a_3\mathbf{k}$$
$$\mathbf{b} = b_1\mathbf{i} + b_2\mathbf{j} + b_3\mathbf{k}$$
$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$
$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

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

Intersecting triangle primitive: intersecting plane

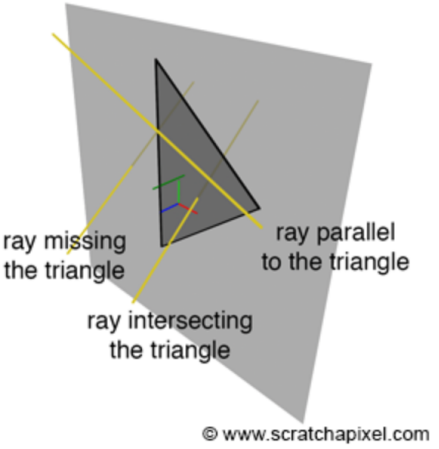
- Intersected point in somewhere on ray:
  - `Phit = P(t) = O + t * R`
- Plane equation :
  - `Ax + By + Cz + D = 0`
  - `D = -(Ax + By + Cz)`
  - `A, B, C` are coordinates of plane normal `N = (A,B,C)`
    - `N` is calculated using triangle vertices
  - `D` is distance from origin `(0,0,0)` to the plane
    - D can be calculated using any triangle vertex: `D = dotProduct(N, v0) = -(N.x * v0.x + N.y * v0.y + N.z * v0.z);`
  - `x, y, z` are coordinates of point on a plane
- Substitute ray equation into plane equation:
  - `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)`
  - Solving by t :

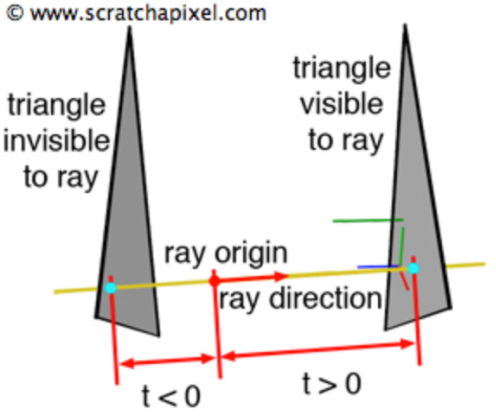
$$t = -\frac{N(A,B,C) \cdot O + D}{N(A,B,C) \cdot R}$$



Intersecting triangle primitive: intersecting plane

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





Intersecting triangle primitive: point inside triangle?

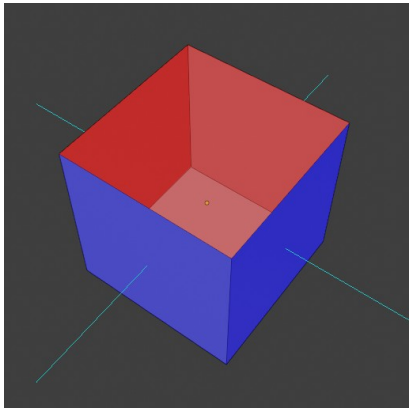
- We have found intersection point P. Is it inside triangle?
- **Inside-out test**
  - `vector3 edge0 = v1 – v0;`
  - `vector3 edge1 = v2 – v1;`
  - `vector3 edge2 = v0 – v2;`
  - `vector3 C0 = P – v0;`
  - `vector3 C1 = P – v1;`
  - `vector3 C2 = P – v2;`
  - `bool q1 = dotProduct(N, crossProduct(edge0, 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;`

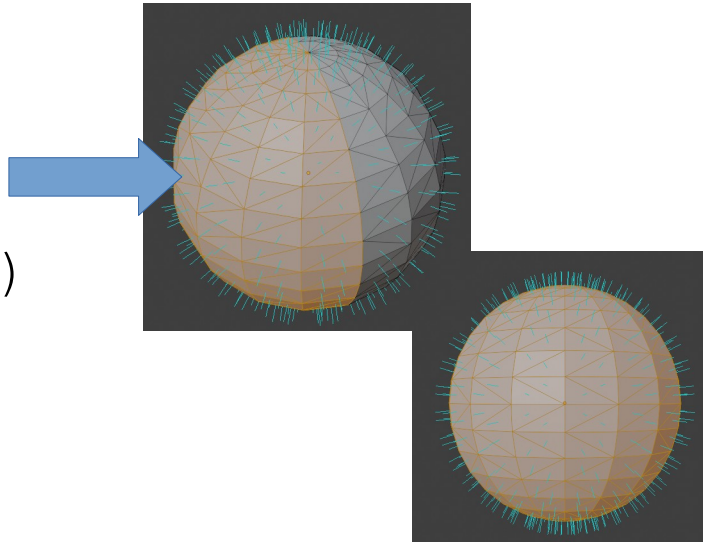
To test if P is inside triangle:

- 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

Single and double sided triangle

- Winding order of triangle vertices defines normal orientation → surface orientation
  - Surface normal pointing outside → **outside surface**
  - Surface normal pointing inside → **inside surface**
- Surface or triangle is **front-facing** if facing the camera, otherwise is **back-facing**
- **Single-sided** primitives: only front-facing
- **Double sided** primitives: both front- and back-facing
- **Back-face culling**: back-facing triangles will not be rendered (drawn)
  - For casting shadows, back-face culling can not be used.
  - Test: `dotProduct(ray.direction, N) > 0`





Testing intersections: trace function

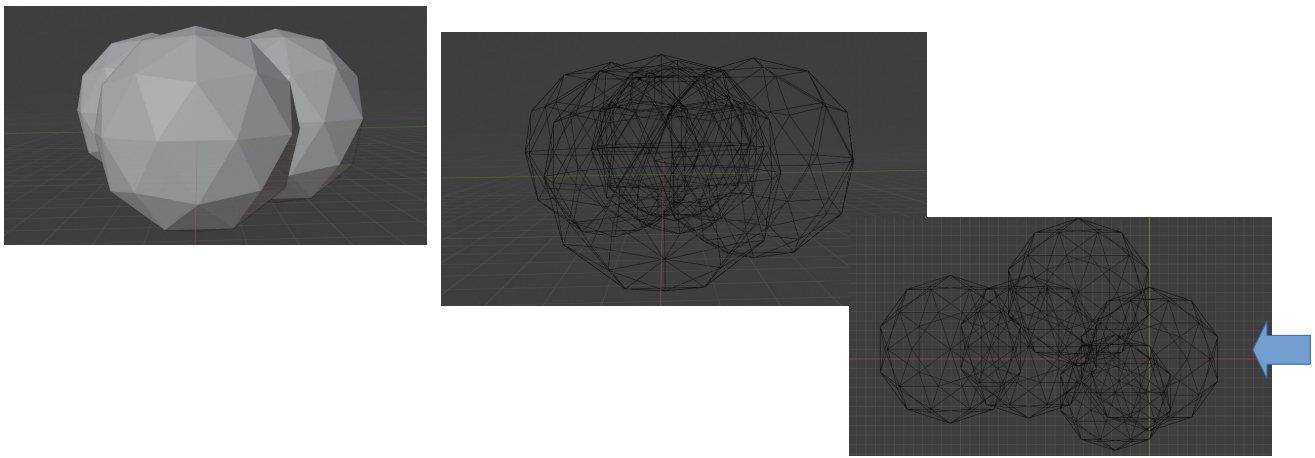
```
bool trace (Ray ray, Object objects, &objIdx, &triIdx)
{
    bool intersect = true;
    for (int k = 0; k < objects.nObjects; ++k)
    {
        for (int n = 0; n < objects[k].nTriangles; ++n)
        {
            if(rayTriangleIntersect(ray, objects[k].triangle[n])
            {
              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

- Inner loop responsible for testing intersections with all objects, that is triangles, can be encapsulated into trace() function
- trace() can be used for any kind of tracing ray into the scene since only information is current ray and objects in the scene:
  - Camera rays used to determine what is visible from camera
  - Later in shading for light transport

Testing intersections: trace function, depth

```
bool trace (Ray ray, Object objects, &objIdx, &triIdx, &tNearest)
{
    bool intersect = 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)
            {
                intersect = true;
                objIdx = k;
                triIdx = n;
                tNearest = t;
            }
        }
    }
    Return intersect;
}
```

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



Trace function, depth

TODO

- Trace function returns the closest intersection, e.g., primary ray with object in the scene.
  - This is fine for **opaque objects** – we are not interested what is behind the intersected surface
- Nevertheless, we will be able to see through transparent objects as well
  - This computation is performed in shading step.
  - In shading step, additional rays for computing incoming light to intersected point are created, depending on object material.

Trace function: intersection context

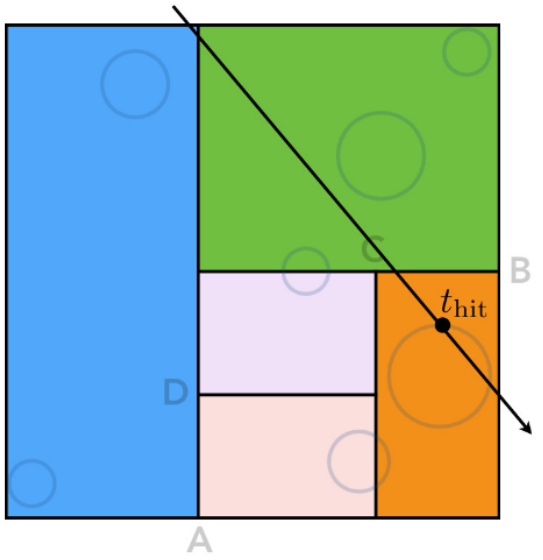
- Trace function returns `objIdx, triIdx` and `tNearest`
  - This is needed to calculate intersection context used in shading:
    - **intersection point**,
    - **normal**,
    - **texture coordinates**, etc.
- Trace function can also directly calculate intersection context – depending on rendering engine design

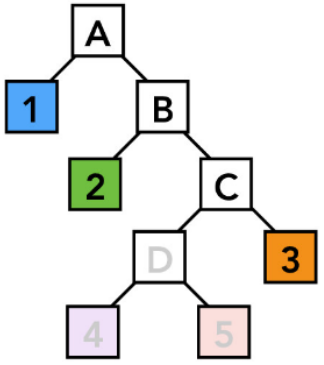
Testing intersections

- Time to render a scene is (directly) proportional to the number of triangles in the scene.
- For shading purposes, all triangles must be stored into memory and each must be tested for intersection for ray
  - Rasterization-based rendering discards triangle not visible from camera (e.g., back faces of object – back face culling) thus faster, but therefore has lower shading capabilities
- Therefore, efficient ray-object intersections are needed!

Testing intersections

- If certain parts of the scene are not relevant for current ray, they can be skipped
  - **Spatial acceleration data structures**
    - object shape information is "sorted" in 3D scene and scene is spatially subdivided.
    - Rays first test larger volumes of the scene and if that volume is not relevant, all objects inside do not have to be tested for intersections!
    - Such acceleration structures require **pre-computation overhead**:
      - Static scenes: only once before rendering.
      - Animated scenes, these datastructures must be pre-computed each time objects move (each frame) before rendering.



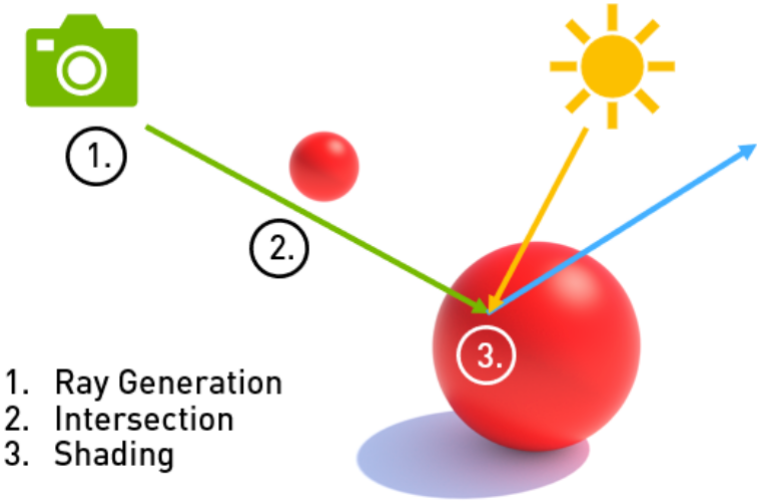


Intersection found

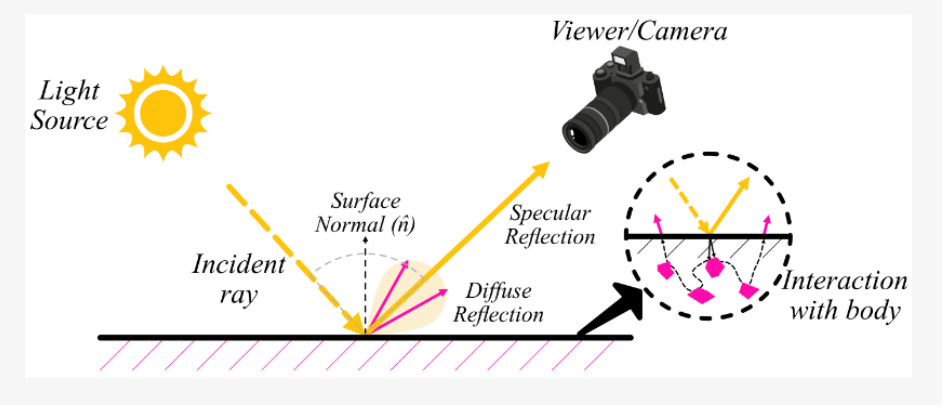
https://cs184.eecs.berkeley.edu/sp22/lecture/10/ray-tracing-acceleration

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/

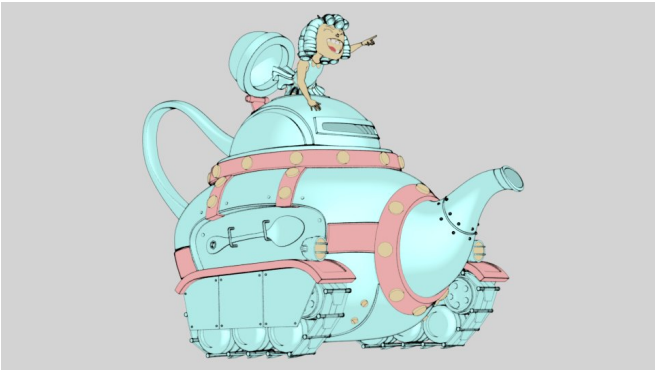


https://www.mdpi.com/1424-8220/22/17/6552

(Non-)Photo-realistic rendering

- Level of photo-realism depends on reproducing the color and intensity of objects (appearance)
- Shading plays crucial role for photo-realistic rendering
- Shading, can be also utilized to render expressive or artistic images → 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

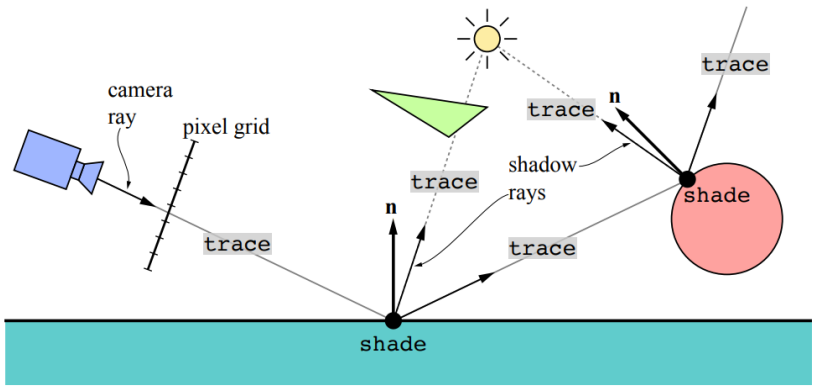
https://www.artstation.com/ar

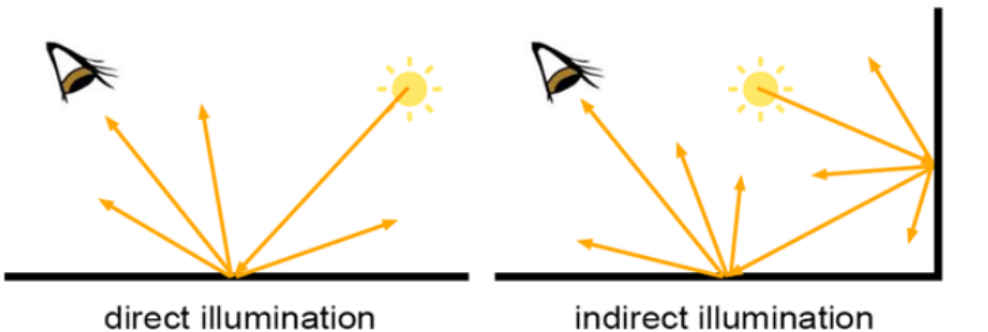


"Rolling Teapot" - Model by Brice Laville, concept by Tom Robinson, render by Esteban Tovagliari - RenderMan 'Rolling Teapot' Art Challenge: https://appleseedhq.net/gallery.html#https%3A%2F%2Fappleseedhq.net%2Fimg%2Frenders%2Frolling-teapot.jpg

Shading and light

- Light is essential to shading
- Incoming light to shaded surface point can be:
  - **Direct illumination**: light emitted directly by light sources
  - **Indirect illumination:** light reflected from another object surface

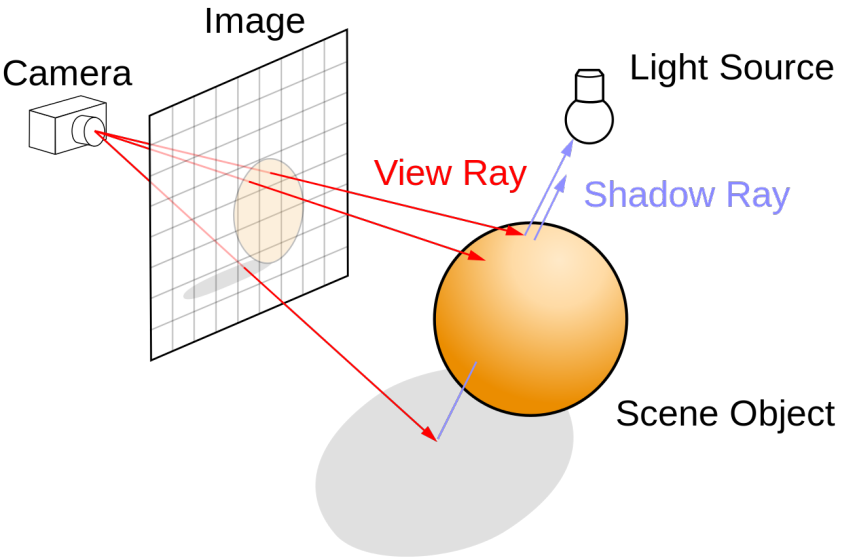




https://www.realtimerendering.com/Real-Time_Rendering_4th-Real-Time_Ray_Tracing.pdf

Direct illumination

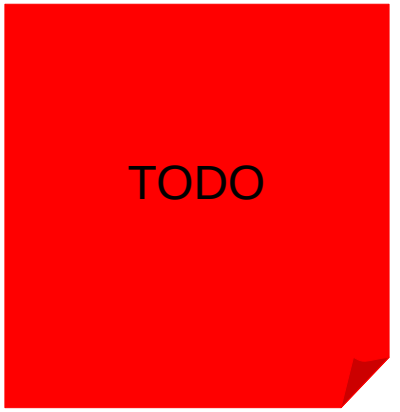
- **Directional lights**
  - Light direction is given as vector L
  - Light color and intensity
- **Point lights or spot lights**
  - **Shadow rays** are generated in from shading point `P` to light source `S` (visibility solving via raytracing):
  - Light direction: `L = normalize(S - P);`
  - Light color and intensity
    - Inverse square law: intensity falls with squared distance to light source



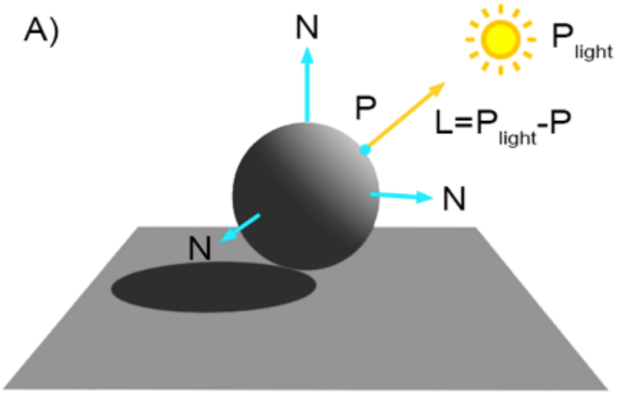
https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

Direct illumination

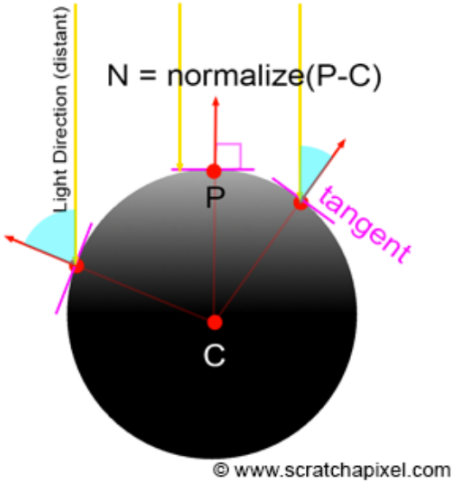
- Area lights

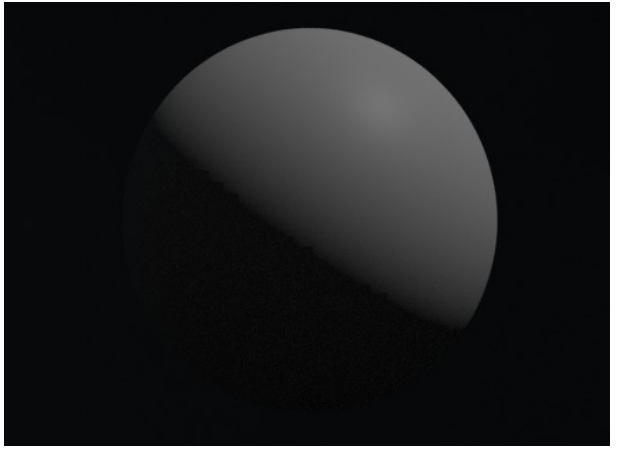


Shading: normals



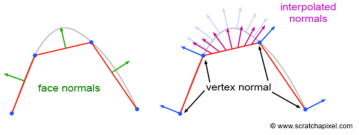
- Surface normal is essential to shading
- Surface normal `N` in shading point defines surface orientation
  - Surface normal and light direction determine surface brightness
  - Simple shading:
    - `shaded_color = surface_color * light_color * light_intensity * max(0, dotProduct(N, L))`



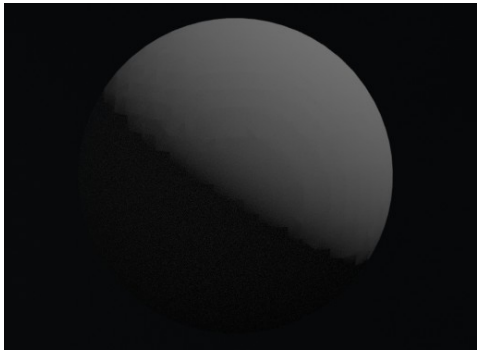


Shading: normals

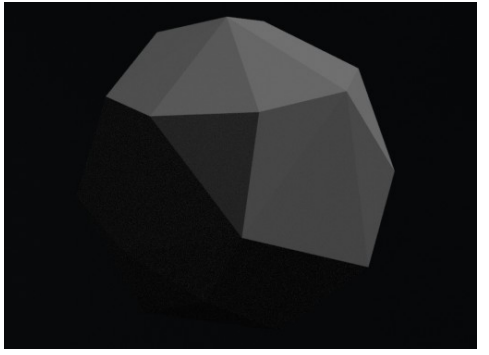
- Triangulated mesh can't represent perfectly smooth surface
- Shading depends on surface normal and triangulated mesh gives faceted look
  - **Flat shading**
- Solution: Gourad shading
  - Produce smooth shading over surface by using **linearly interpolated vertex normals**
  - Linear interpolation of vertex normals is done using barycentric coordinates
- Vertex normals are:
  - Computed on the fly
  - Given from smooth surface from which triangulated mesh was created



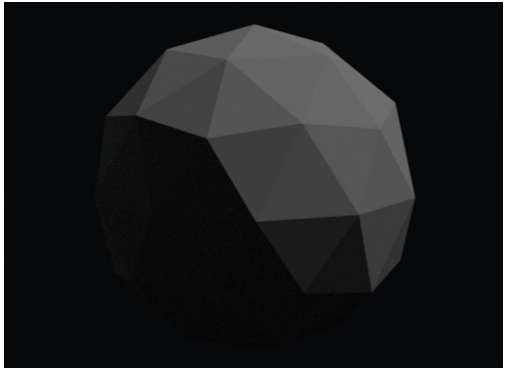
Implicit sphere

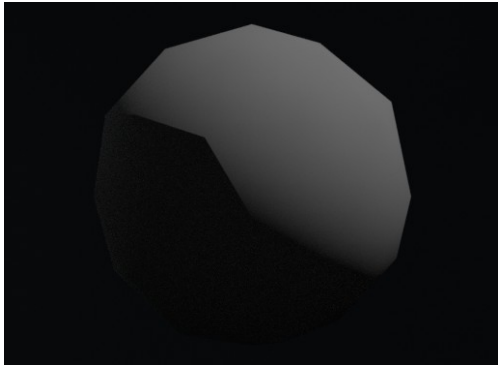


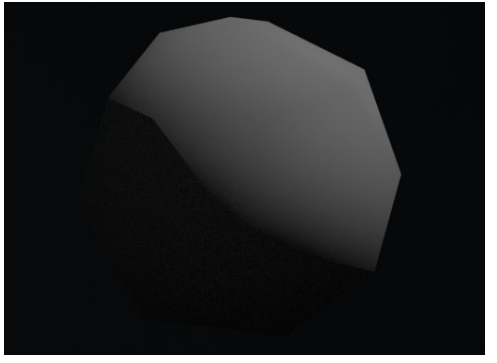
Triangulated mesh from implicit sphere



Triangulated mesh from implicit sphere with smooth shading

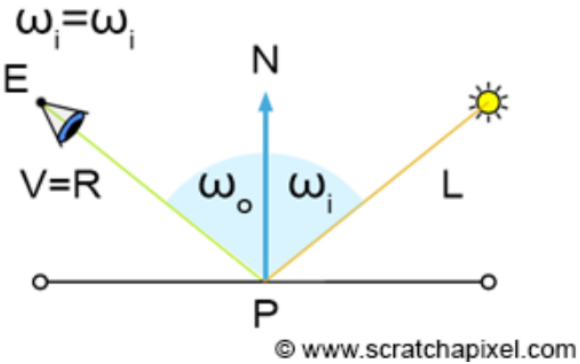






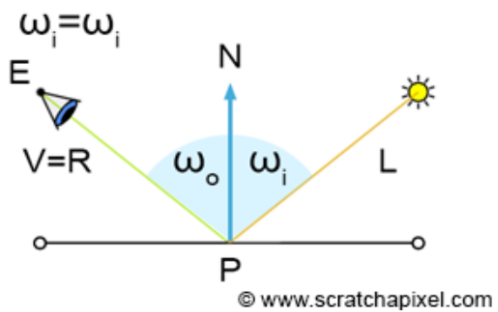
Shading: material

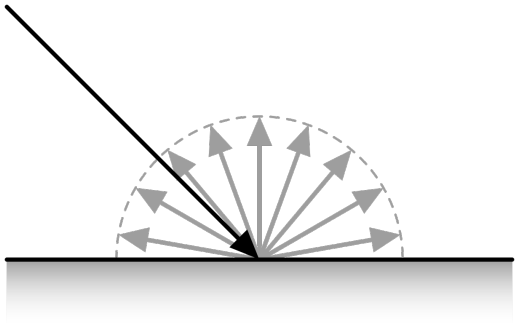
- Appearance of object surface is further defined by reflectance model – **BRDF** `f(L,N,V)`
  - Mathematical model approximating interaction of light and microscopic structure of object material
  - Besides parameters, it depends on incident light direction `L` and view direction `V`
  - Diffuse/Lambertian, specular, Phong, Blinn-Phong, Lafortune, Torrance-Sparrow, 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))`

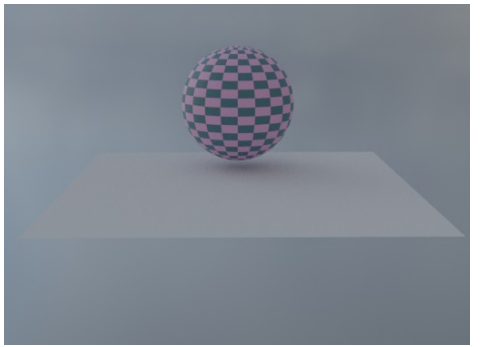


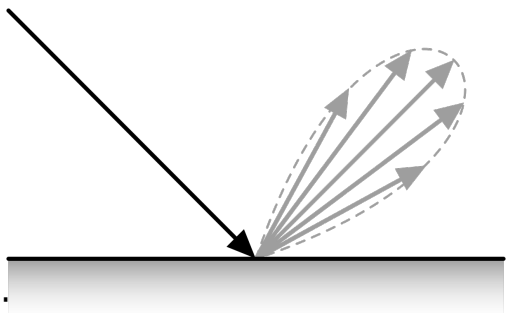
Shading and BRDF

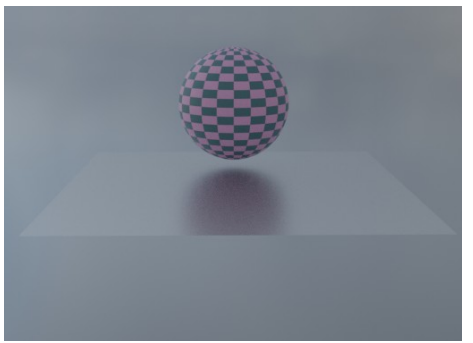
- **Amount reflected light in view direction `v` given indicent direction** `L` - BRDF: `f(V,N,L)`
  - L – incoming/incident direction
    - Angle of incidence: ω_i (between L and N)
  - V – view direction
    - Line joining eye/camera E and shaded point
  - R – outgoing/reflected light direction
  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

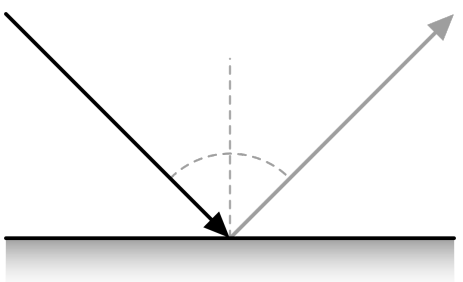


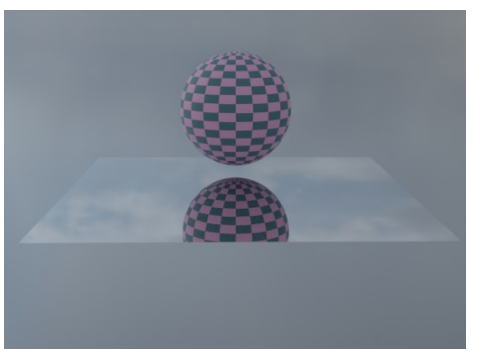






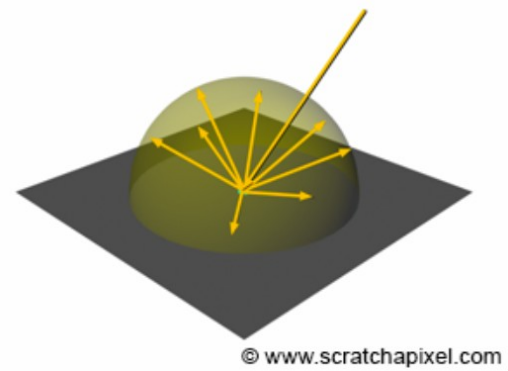


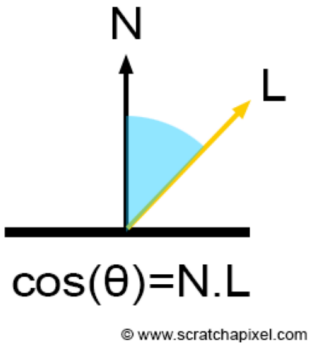




Shading: diffuse/Lambert surface

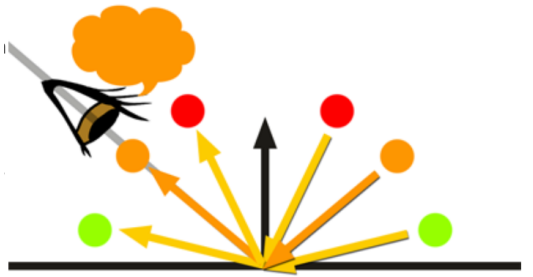
- Light falling on surface is attenuated based on surface normal and incoming light direction
  - Lambert's cosine law: `dotProduct(N, L)`
- Diffuse BRDF: `albedo / PI`
  - `albedo RGB in [0,1]`
  - `Albedo = (reflected_light / incident_light)`
- Diffuse surfaces are reflecting light in all directions equally – over 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**

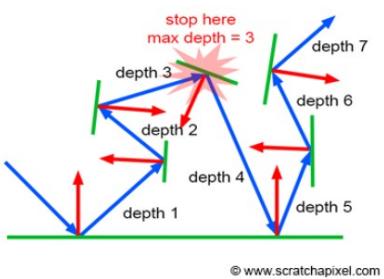




Shading: perfect specular surface

- Light bounces in direction symmetrical to incident direction around normal at shading point → **law of reflection**
  - `incident_angle = reflected_angle`
- Reflected direction:
  - `R = L – 2 (N * L) * N`
- 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!





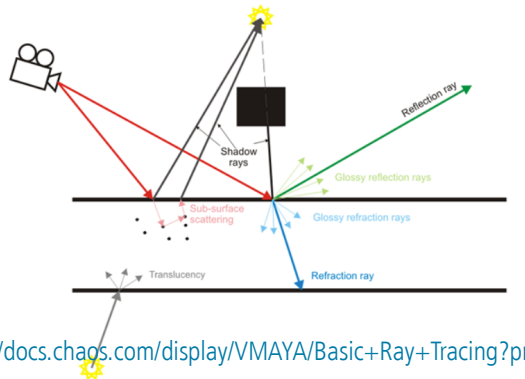
Shading: perfect specular surface

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

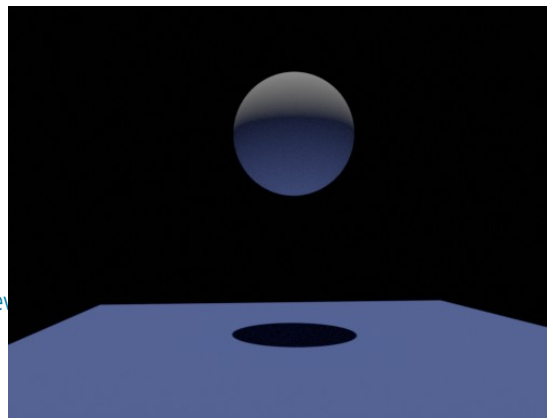
Without objects in 3D scene, specular sphere is not visible since it reflects only black color.

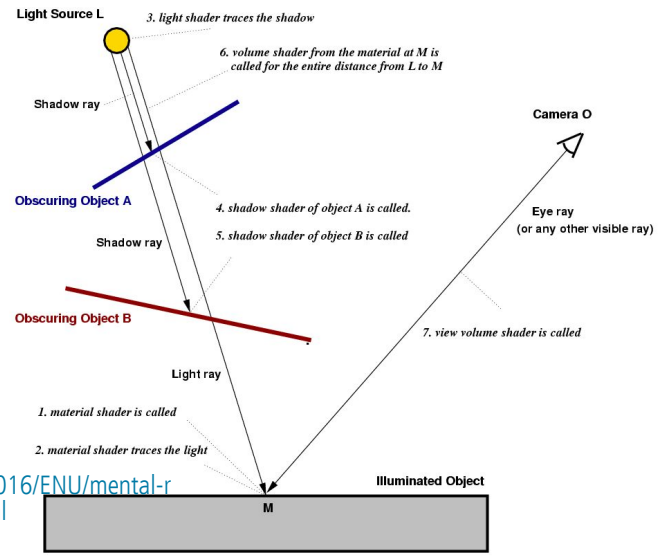
Shading: shadows

- After primary ray intersection has been found, shadow ray is sent from shading point to light source.
- Beside information of light direction, intensity and color, the purpose of shadow ray is to determine if some objects are intersected on the path
- If shading point is shadowed, it has zero light contribution → black color



https://docs.chaos.com/display/VMAYA/Basic+Ray+Tracing?prev





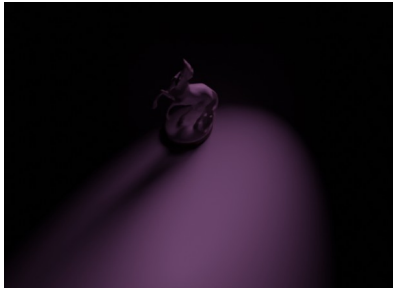
https://help.autodesk.com/cloudhelp/2016/ENU/mental-ray-docs/mr_docs/manual/node102.html

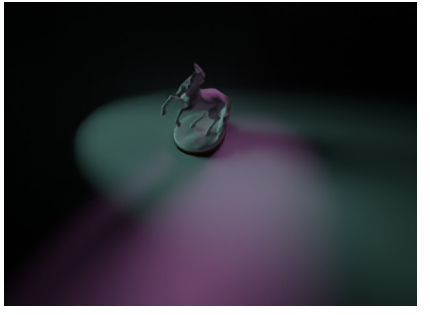
Shading: Multiple lights

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

$$shading\_result = \sum_{i=1}^{n} f(l_i, n, v) c_{light_i} (n \cdot l_i)$$

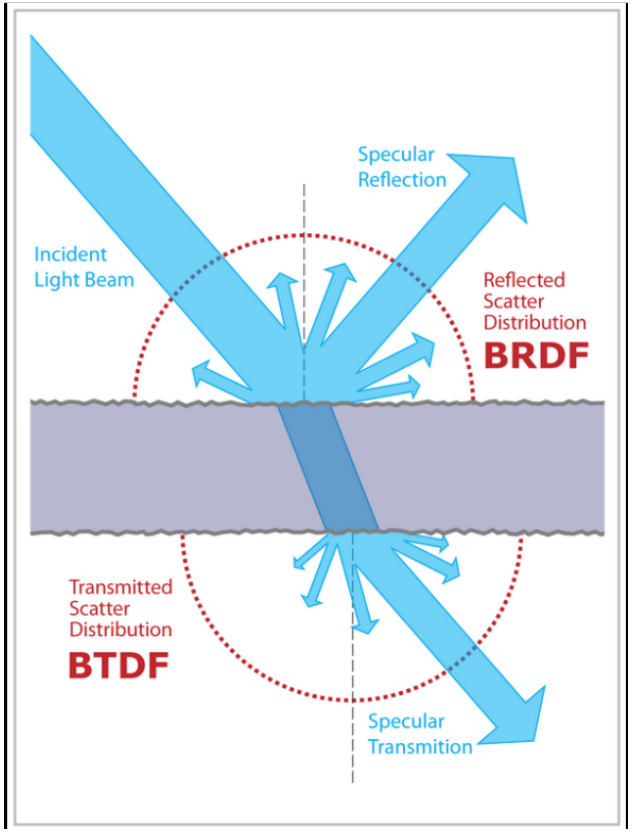
$$c_{light_i} = light\_color_i * light\_intensity_i$$





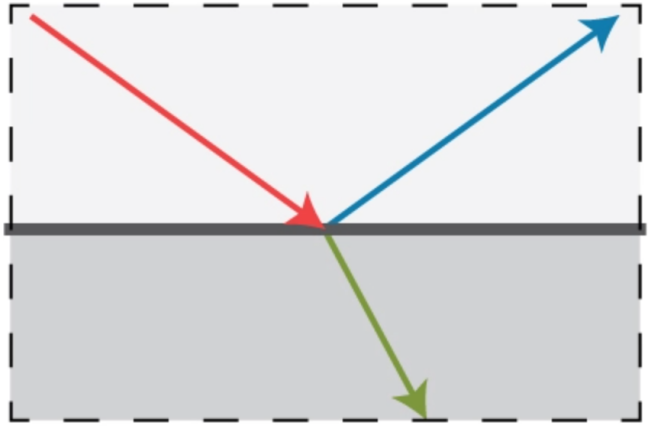
Shading and BTDF

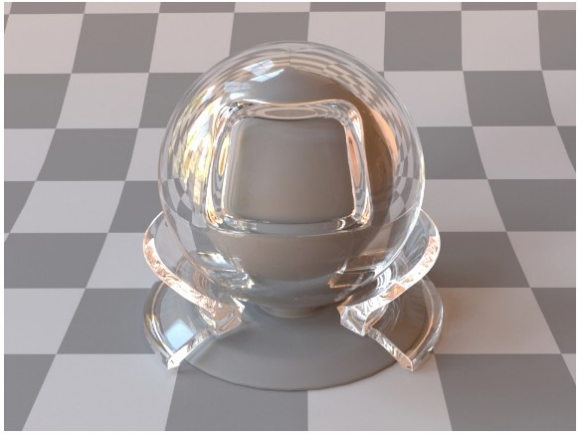
- For transparent objects (e.g., glass), light is reflected and **refracted**
  - Reflection: light changes direction traveling outside of the surface
  - Refraction: light changes direction traveling inside of the surface
  - Transmission: light that enters object on one side and leaves object on other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



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





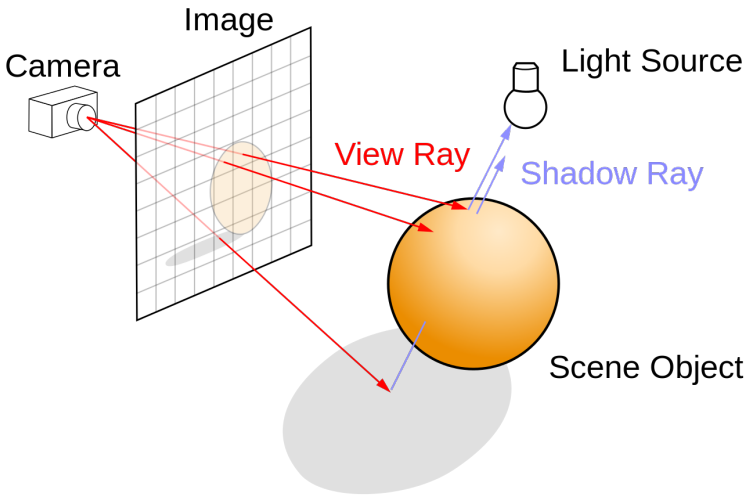
Shading and perfect specular transmission

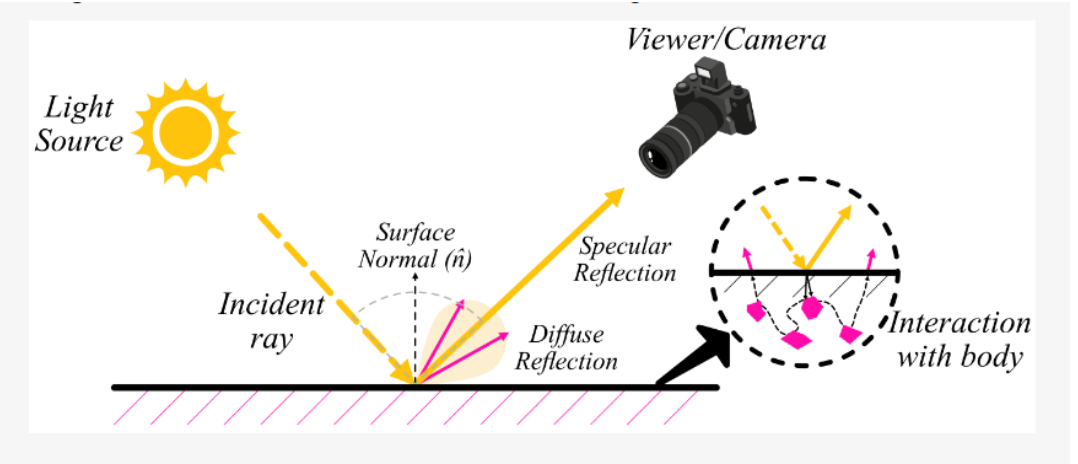
- Perfectly specular surfaces only reflect and transmit other surfaces in the 3D scene

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

Direct illumination recap

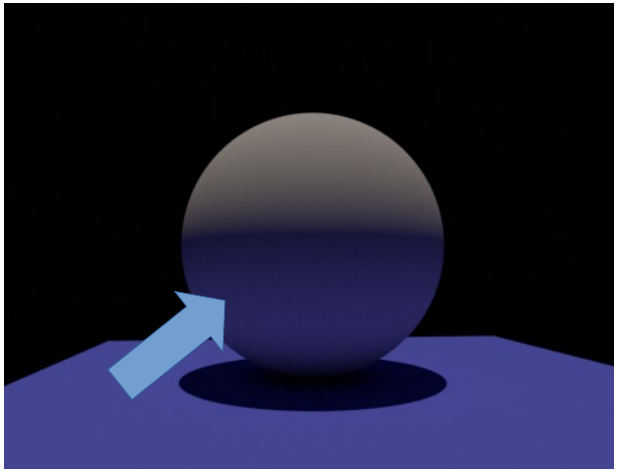
- For now we have:
  - Created primary ray
  - Traced primary ray for intersection
  - Traced shadow ray from shading point to light source
  - Calculate shading based on surface shape (normal), material, primary ray and shadow ray information



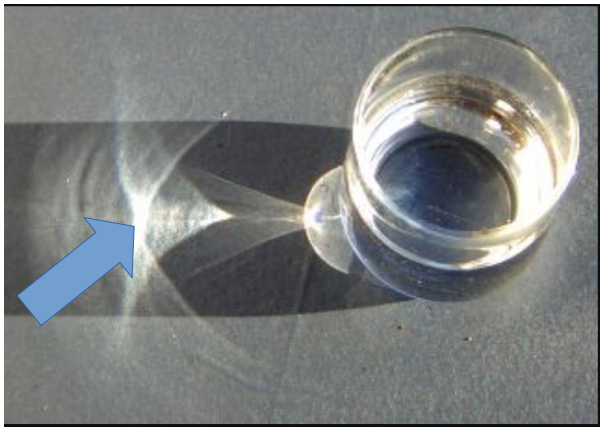


Indirect illumination

- Shading very much depends on incoming light
  - Light coming directly from light source
  - Light can travel from light source, reflecting from objects and finally fall on shaded surface
    → **indirect illumination**
- Gathering indirect light on shading point is called **light transport**



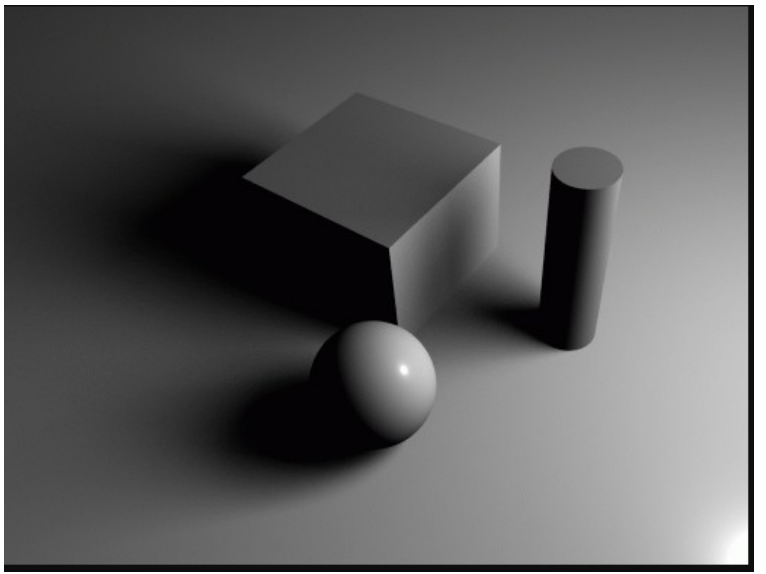
Indirect diffuse – diffuse objects reflect light which illuminates other objects in the scene



Indirect specular - specular objects reflect light which illuminates other objects in the scene.
https://upload.wikimedia.org/wikipedia/comm

Indirect illumination

- Besides material properties, geometrical shape also contributes to appearance of the scene:
  - **Soft shadows**



https://renderman.pixar.com/resources/RenderMan_20/softShadows.html

Shading and light transport

- Appearance of object and can be divided into:
  - **Shading**: how object appears.
    - 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

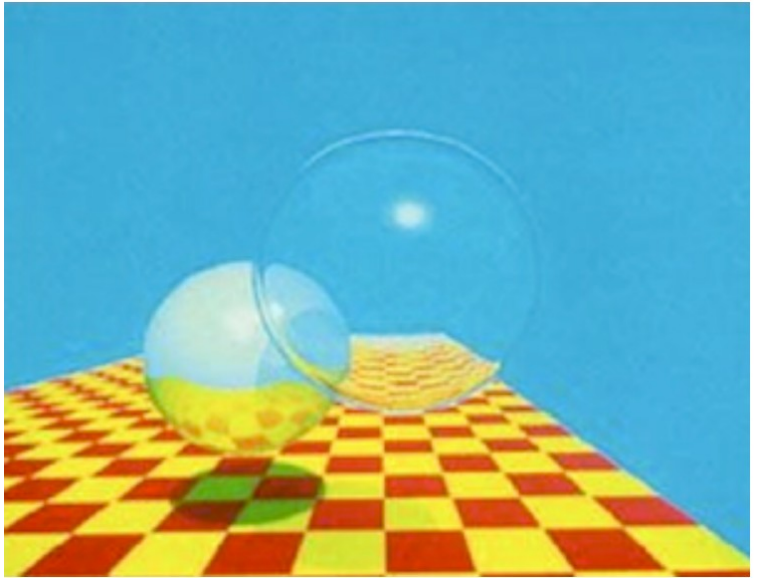
- **Global illumination:** simulating both direct and indirect illumination
- This problem is described by **reflectance equation:**
  - Color in intersected point L_o is results of summing all incoming light from hemisphere of directions **ω**, multiplying it with BRDF f(l,n,v) and with max(0, dotProduct(n,l))

$$L_o(p,v) = \int_{l \in \omega} f(l,n,v) L_i(p,l)(n \cdot l)^+ dl$$

- Which is a special case of **rendering equation**
- Solving rendering equation requires advanced light transport strategies

Global illumination and light transport

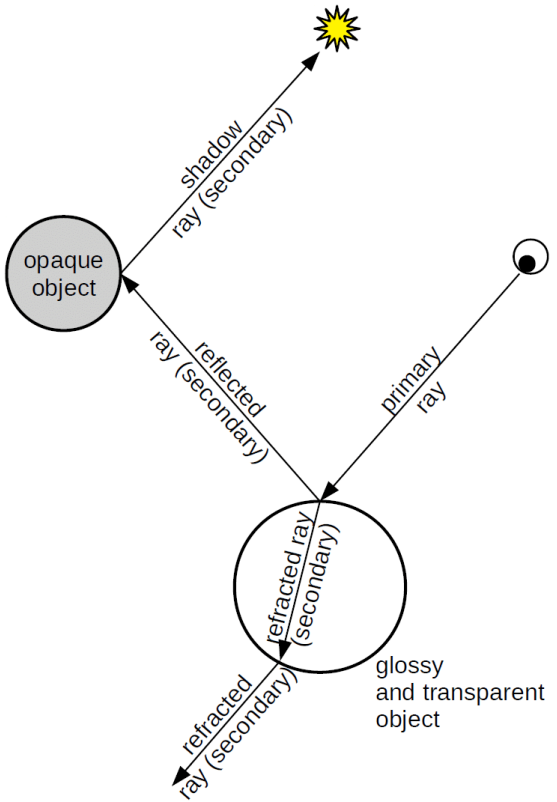
- Various light transport strategies based on ray-tracing for solving global illumination exist. Most popular:
  - **Path tracing** and **bidirectional path tracing**
- Classical example of light transport is **Whitted ray-tracing**
  - This strategy is not solving global illumination
  - This strategy is utilizing ray-tracing for introducing indirect illumination
  - Advanced light transport strategies build on this method



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

Whitted ray-tracing

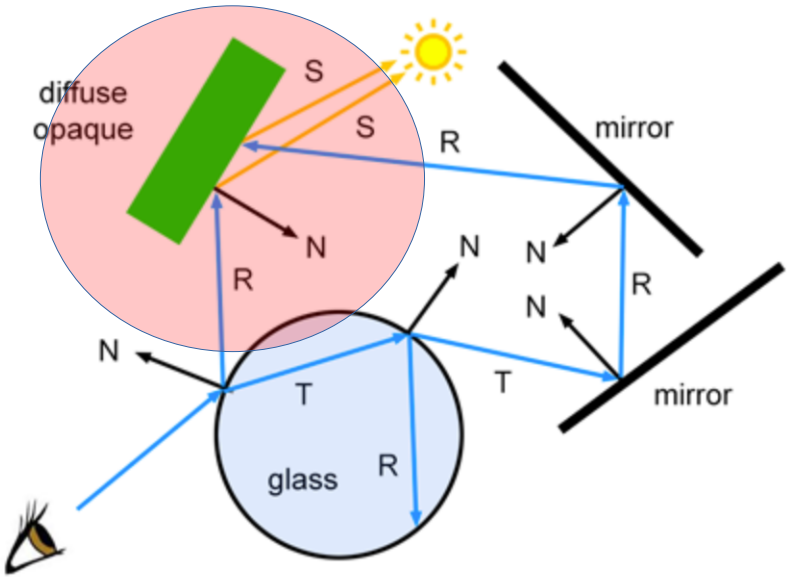
- Simulating complex reflections and refractions
- Use scattering models (BSDF) to compute light paths
  - Laws of reflection and refraction are used to compute direction of light rays intersecting reflective or transparent surfaces → **secondary rays**
  - Follow light rays bouncing around the scene and find out color of objects they intersect
- 3 main cases for light paths



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

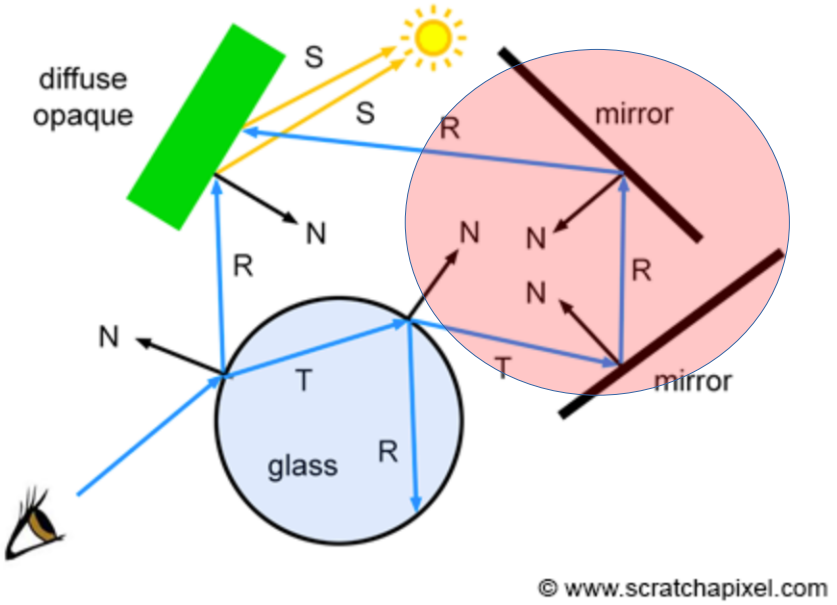
Whitted ray-tracing: case 1

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



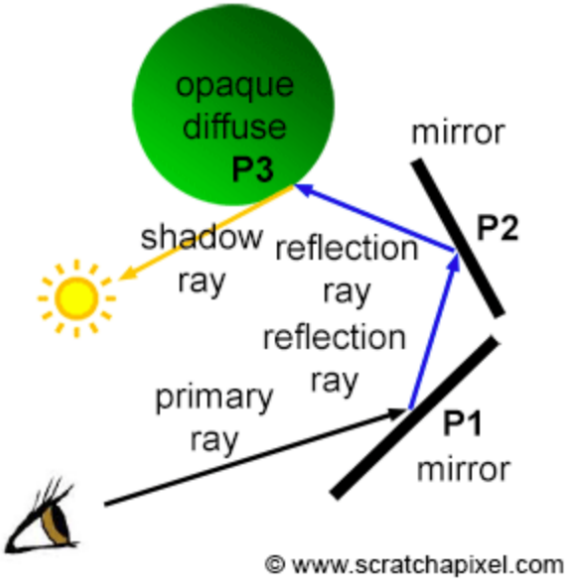
Whitted ray-tracing: case 2

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



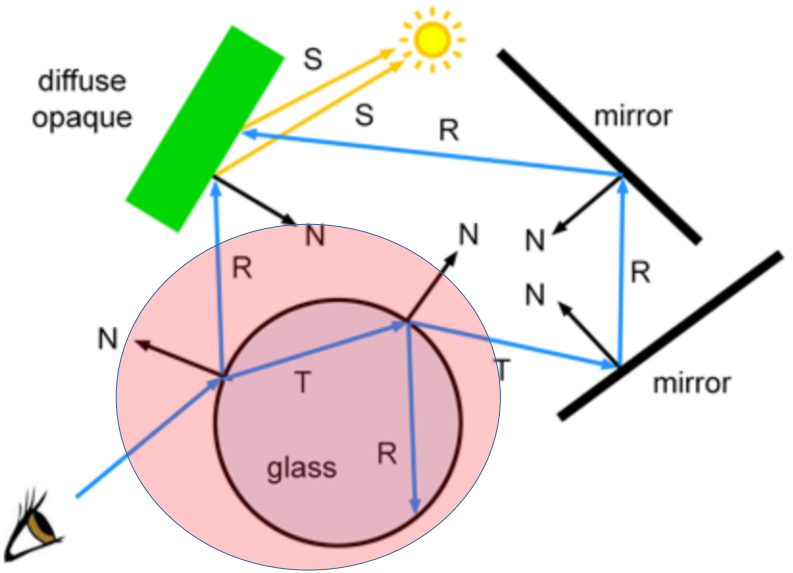
Whitted ray-tracing: reflection ray

- 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



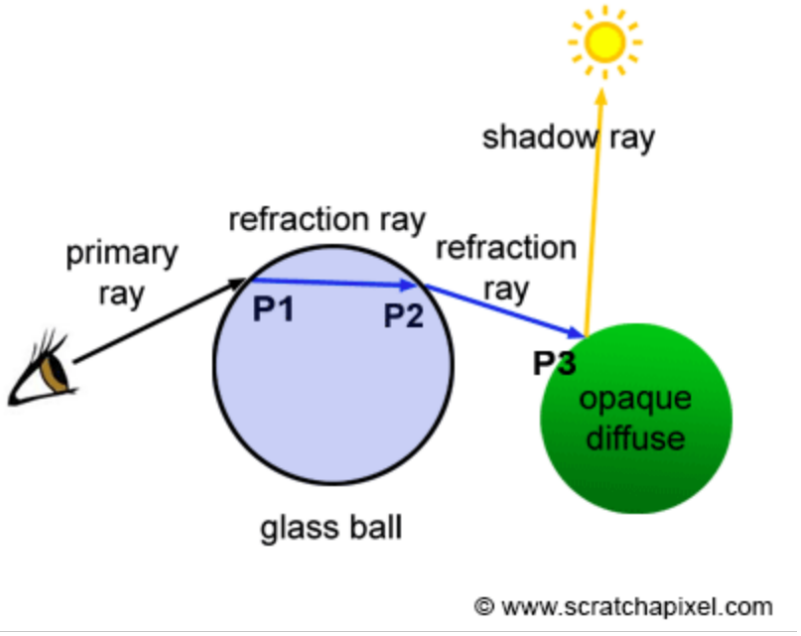
Whitted ray-tracing: case 3

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



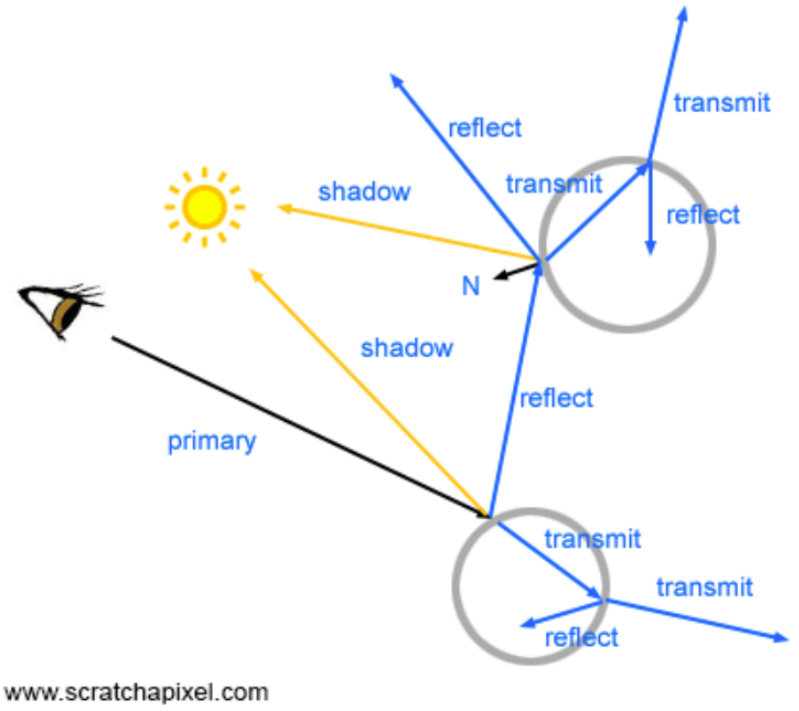
Whitted ray-tracing: refraction ray

- 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



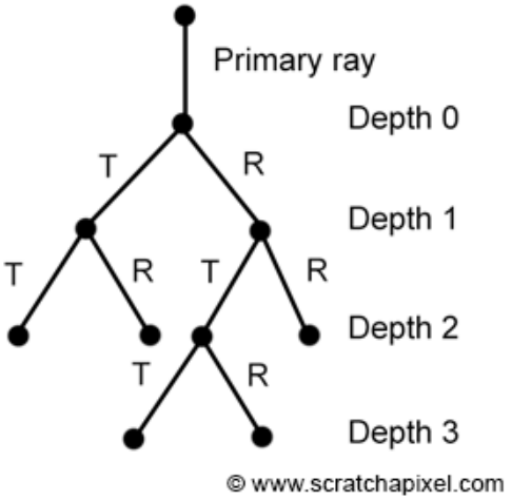
Whitted light-transport properties: recursive nature

- 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



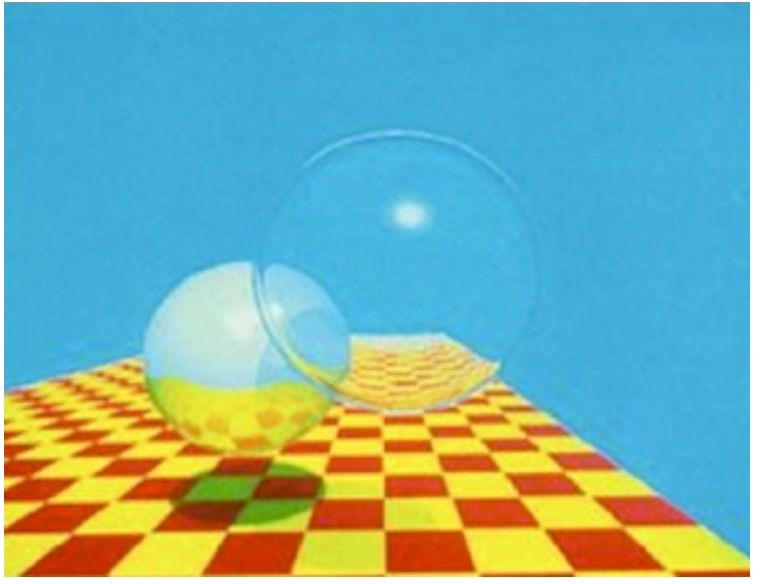
Whitted light-transport properties: tree of rays

- 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



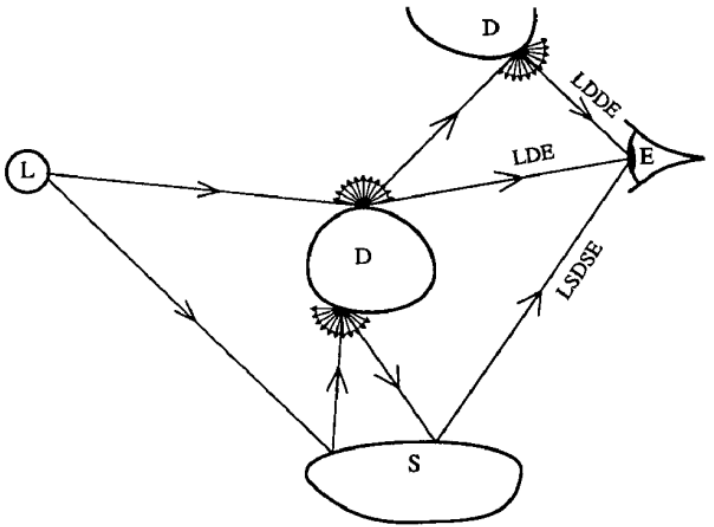
Whitted ray-tracing and light transport

- 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

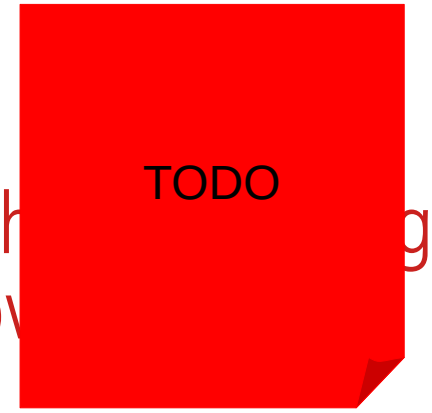
Whitted ray-tracing and beyond

- 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-tracing method
  - Correct solution which generates global illumination
  - After camera ray intersection is found and during shading evaluation,may rays are generated in different directions
    - 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

Practical raytracing-based rendering

Code

Code



- https://www.scratchapixel.com/code.ph gin=/le ssons/3d-basic-rendering/ray-tracing-ov

- https://raytracing.github.io/books/RayTracingInOneWeeke nd.html#overview
- https://www.scratchapixel.com/lessons/3d-basic-rendering /ray-tracing-overview/light-transport-ray-tracing-whitted
-

Ray-tracing based rendering: verdict

- Good:
  - Ray-tracing based rendering is clear and straightforward method to implement*
  - It represent unified framework for computing what is visible from camera (primary ray intersections), with inherent perspective/orthographics projection and computing light transport – visibility between surfaces in 3D scene (secondary ray intersections) required for the shading process.
  - Powerful shading capabilities when it comes to light transport

* 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 are often very expensive
  - For advanced light transport needed in shading stage, ray-tracing-based renderer must store all scene objects must be readly available for intersection testing and material evaluation*. An example where is this needed is 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.

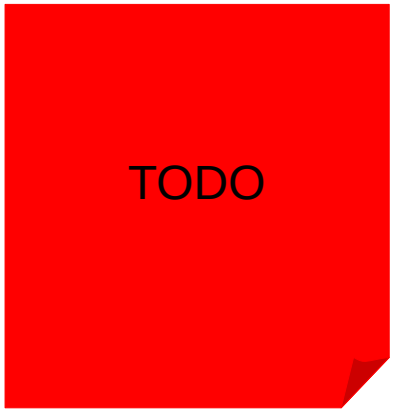
* As we will discuss, this is not needed for rasterizer since only visible geometry is taken in account – which on the other hand disables advanced light transport (again, visibility calculation between objects in the scene) and thus shading.

Practical-raytracing based rendering

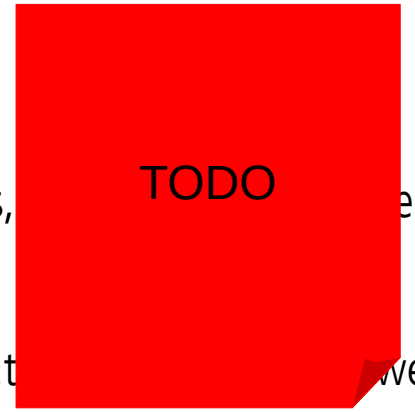
Production

Back to motivation scene

- Show cycles rendering in Blender
- TODO



Note: real-time ray-tracing



- Due to nature of ray-tracing-based visibility and shading calculations, [redacted] en for CPUs
- Porting ray-tracers to GPU was done in 90s
- Ray-tracing ideas are often used on GPU and rasterizer renderers, but [redacted] we discussed
  - Hybrid approaches exists: rasterization might be used to determined objects/surfaces visible from camera. Raytracing may be employed for reflection and refraction calculations.
- Newer graphics hardware (currently led by Nvidia with RTX family of graphics cars) and new graphics APIs (DirectX 12, Vulkan) are supporting hardware-accelerated ray-tracing – a new hardware and then software element called raytracing kernel can be now used in combination with rasterizer-based renderer to compute complex scene effects such are reflections, soft shadows, indirect illumination, etc.

Acknowledgments

- https://www.scratchapixel.com