Foundations of Ray-tracing

Syllabus

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

- Ray-tracing based rendering
  -

- <TODO: image rendered with ray-tracing based rendering engine>

Raytracing-based rendering overview

Introduction

- 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.
- Ray-tracing is a method inspired by physics of light → realistic image synthesis
- <TODO: image rendered with ray-tracing based rendering showing these effects>

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

Introduction

- Rendering process: visibility solving and shading
- Ray-tracing, fundamentally, **solves visibility problem**: two points are visible to each other if line segment that joins them does not intersect any obstacle.
  - First, we need to determine which objects are visible from camera – visibility problem! Based on defined camera, ray-tracing-based rendering is inherently providing **perspective or orthographic projection** in rendered image.
  - Secondly, we need to determine color and intensity of visible objects. This is solved using **shading** which relies on **light transport** (which relies on visibility problem) to obtain incoming light to surface.

Three steps of ray-tracing-based rendering

- Rendering: assign a color to each pixel of the frame
- **Generate ray** for each pixel of film plane → camera ray
- **Ray-geometry 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
  - Requires light-matter calculation
  - Requires light transport for gathering incoming light

Ray-tracing-based rendering: trace & shade

- Raytracing can be described with two functions:
  - **trace()**
  - **shade()**
- **trace()** is geometrical part of algorithm and it is responsible for finding closest intersection between ray and the primitives in 3D scene.
- Once intersection is found, **trace()** returns color of the ray by calling **shade()**

Trace() for camera rays

- As the goal of rendering is to find colors of pixel in the image, trace() function is used for rays generated by camera: eye or camera rays
- Now, the goal of trace() function is to find closest intersection of camera ray with 3D scene objects.
- Naive trace() function loops through all n objects in the scene
  - This is slow for complex scene → o(n) performance
  - Spatial acceleration structure (BVH or k-d tree) is used to achieve O(log(n)) performance

shade() for closest intersection

- Color in closest intersection is calculated using shade() it this is done using:
  - Material information of intersected point
  - Incoming light computed using trace() function
- Shade() can be arbitrarily complex:
  - It can just return the color of object
  - It can use material information with incoming light information
    - It can ask trace() for closest light sources
    - It can ask trace() to gather all incoming light
- 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.

trace() for light transport

- Simplest case is that trace() calculate visibility between shaded point and light source
  - Shadow ray: is shaded point in the shadow or visible to light source?
- Trace() can also use normal at intersection and compute reflection or refraction ray
  - Perflectly sharp reflections and/or refractions with sharp shadows is called Whitted ray-tracing.

Digression: ray-casting

- Ray-casting is often used in trace() function for determining visibility between two points
- <image: raycasting>
- It is also very useful operation in different applications
  - Ambient occlusion
  - modeling

Practical foundations

- Overall structure of Whitted ray-tracer which is bases of many other rendering variatns (e.g., path-tracing) can be described with three functions.
- TODO: images - https://www.realtimerendering.com/Real-Time_Rendering_4th-Real-Time_Ray_Tracing.pdf

```
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

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
```

- Shade() function is implemented by user
  - This is were knowledge about materials: scattering functions and textures comes in
- Traversial and intersection testing takes place in trace() function often implemented on CPU but moving to GPU using compute shaders in Vulkan or DXR

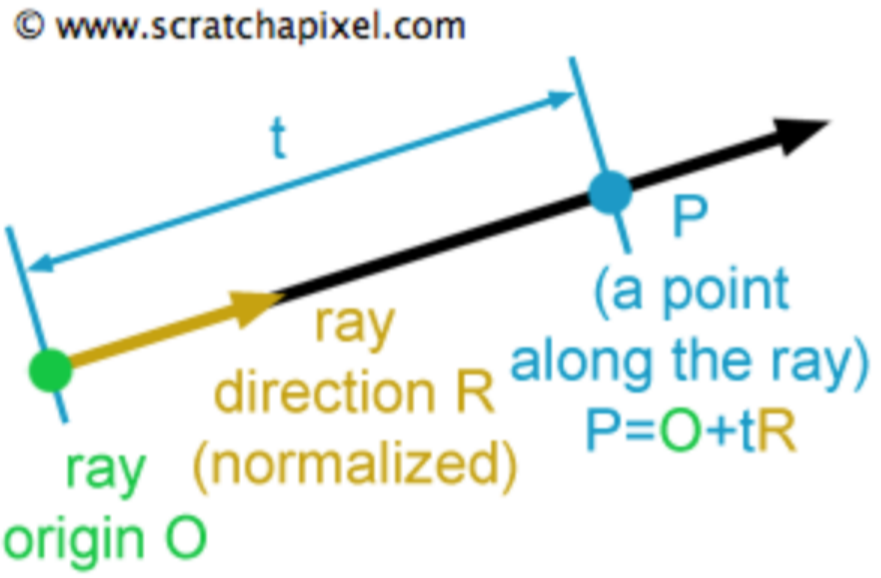
Whitted ray-tracing and beyond

- 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

Whitted ray-tracing

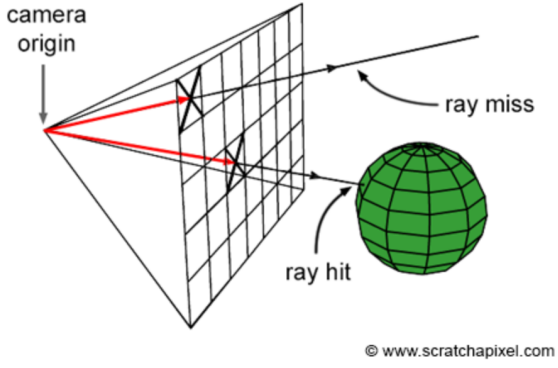
Ray

- Ray is fundamental element of ray-tracing.
  - Used for finding visible objects from camera
  - Used for shading visible objects during light transport.
- Ray is defined as:
  - `vector3f origin;`
  - `vector3f direction;`
- Ray defines **half-line** and any point `P` on it is defined using parametric equation:
  - `P(t) = origin + t * direction;`
  - `t` – distance from `origin` to `P`
  - **`t > 0`** → `P is` in front of ray's `origin`
  - `t < 0` → `P is` behind the ray's `origin`



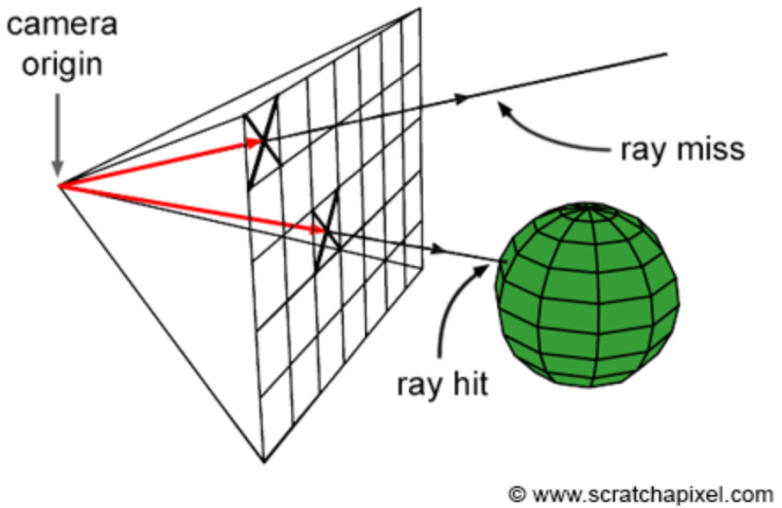
Generating camera rays

- Camera defines:
  - 3D viewpoint: aperture (eye) position and orientation of c
    of sight
  - Field of view: how much of the scene we see
  - Frame: array of pixels
- Rays are generated from camera: **backward/eye-tracing**
  - Rays are generated by starting from camera origin (eye/aperture) and passing through the center of each pixel in the film plane.
- Generated rays are called **camera/primary rays.**
  - These rays will be used to compute the visible objects for current camera position → **ray-casting**



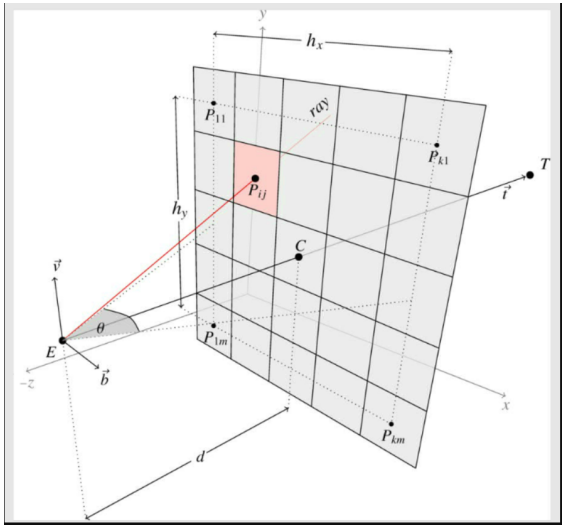
Generating camera rays

```
Vector3f framebuffer[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…
                framebuffer[j * imageWidth + i] = shadingResult;
            }
            if (intersect(ray, object[k])
            {
                // Background hit. Compute background color...
                framebuffer[j * imageWidth + i] = backgroundColor;
            }
        }
    }
}
```



Generating camera rays

- Create primary ray for each pixel of the frame which is used for intersection
  - All objects in 3D scene are defined in **world space**
  - Rays must be expressed in world space for correct intersection computation
- Ray is traced from camera origin to pixel center
  - Camera origin is point in **camera space** → apply **camera-to-world matrix** to obtain world space camera origin
  - Point on a pixel must computed from frame pixel coordinates and transformed in world space using **camera-to-world matrix**
- Frame pixel coordinates are in **raster space**
  - Assume: top left of frame has pixel coordinates (0,0)
- Image plane, where frame must be mapped is in camera space
  - Assume: image plane is located one unit away from camera origin and aligned in negative z axis
  - Assume: image plane is centered around camera origin
- Task: **find relation between pixels in raster space and coordinates of those pixels in world space**

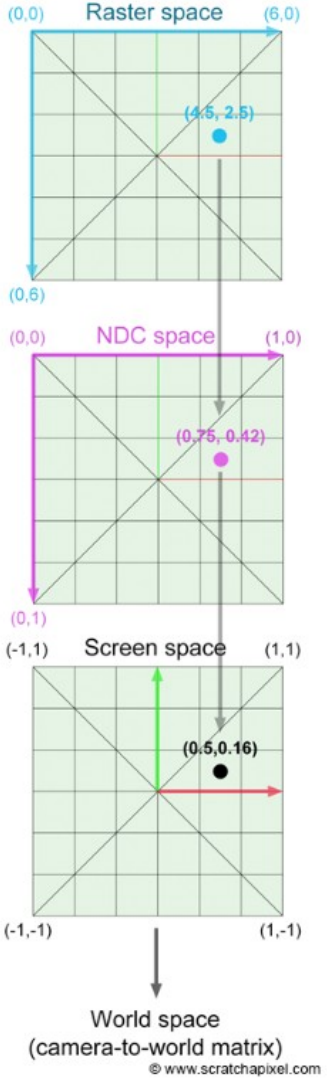


Generating camera rays (square frame)

- Finding pixel center coordinate in world space
  - Normalize pixel position using frame dimensions → **NDC coordinates** (space) [0,1]
  - Remap NDC coordinates to **screen/camera coordinates** (space) [-1,1]

$$PixelNDC_x = \frac{(Pixel_x + 0.5)}{ImageWidth},$$
$$PixelNDC_y = \frac{(Pixel_y + 0.5)}{ImageHeight}.$$

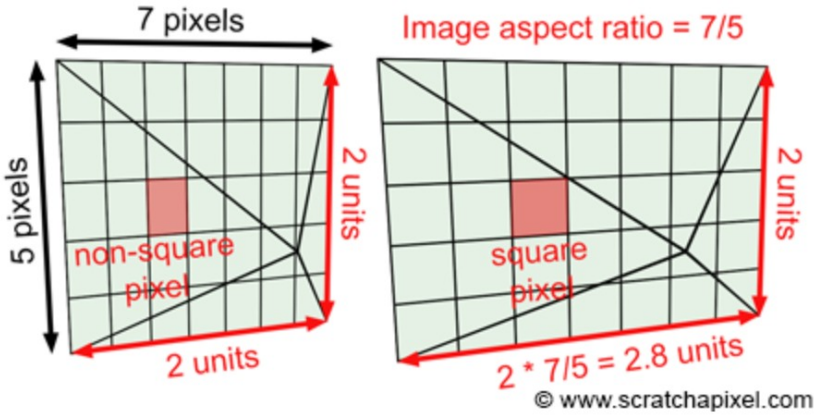
$$PixelScreen_x = 2 * PixelNDC_x - 1,$$
$$PixelScreen_y = 1 - 2 * PixelNDC_y.$$



Generating camera rays (frame with arbitrary aspect ratio)

- Task: Enable frame aspect ratio to be arbitrary
  - Scale image plane by image aspect ratio

$$ImageAspectRatio = \frac{ImageWidth}{ImageHeight},$$
$$PixelCamera_x = (2 * PixelScreen_x - 1) * ImageAspectRatio,$$
$$PixelCamera_y = (1 - 2 * PixelScreen_y).$$



Having more pixels in width causes different height and width lengths of the frame. To make sure that frame pixels are squared, image plane must be scaled by image aspect ratio.

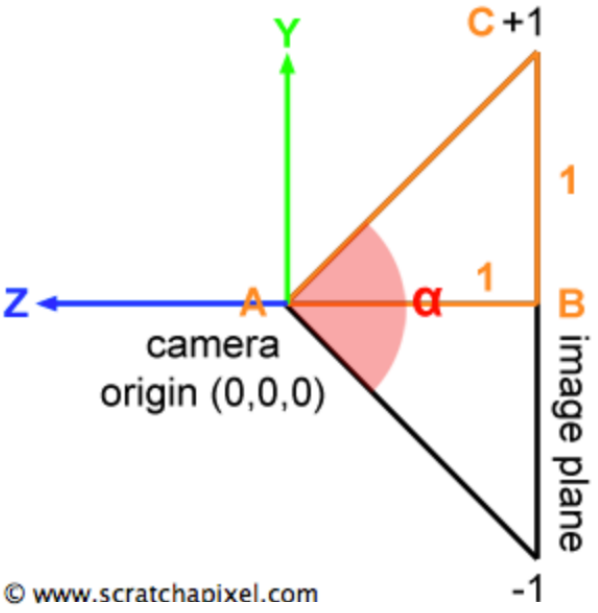
Generating camera rays: field of view

- So far image plane y coordinates are in [-1,1] (screen space)
- We know:
  - image plane is 1 unit from camera origin
  - Height of image plane is 2 (from -1 to 1)
- Triangle joining camera's origin and image (film) plane
  - Angle alpha is field of view
- Camera space pixel coordinates:

$$PixelCamera_x = (2 * PixelScreen_x - 1) * ImageAspectRatio * tan(\frac{\alpha}{2}),$$

$$PixelCamera_y = (1 - 2 * PixelScreen_y) * tan(\frac{\alpha}{2}).$$

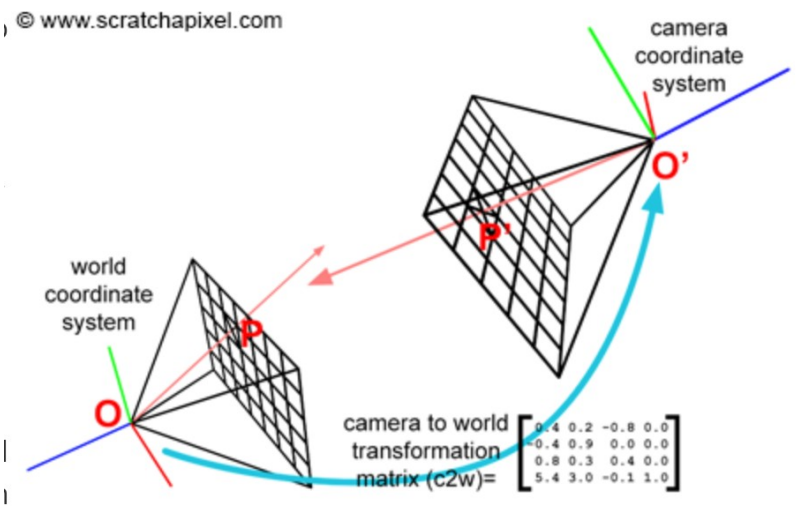
$$P_{cameraSpace} = (PixelCamera_x, PixelCamera_y, -1)$$



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

Generating camera rays: world space

- Now camera origin and position of pixel in image plane of camera are in camera space.
- Camera space ray can be constructed:
  - `vector3 RayOrigin = cameraOrigin;`
  - `vector3 rayDirection = normalize(pixelPosition – cameraOrigin);`
- World space ray can be constructed:
  - `Camera-to-world matrix` is first applied applied on pixel position and camera origin
  - `vector3 RayOrigin = cameraOrigin;`
  - `vector3 rayDirection = normalize(pixelPosition – cameraOrigin);`
- `Camera-to-world` matrix is specified using `look-at` matrix.

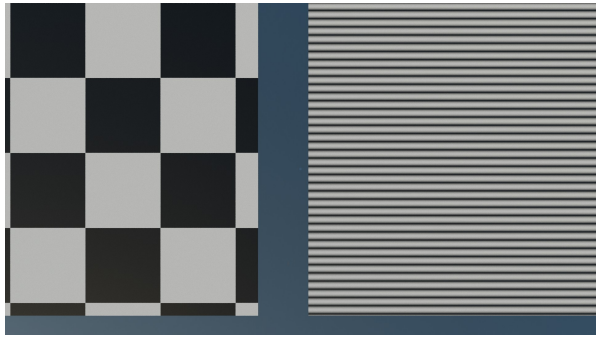


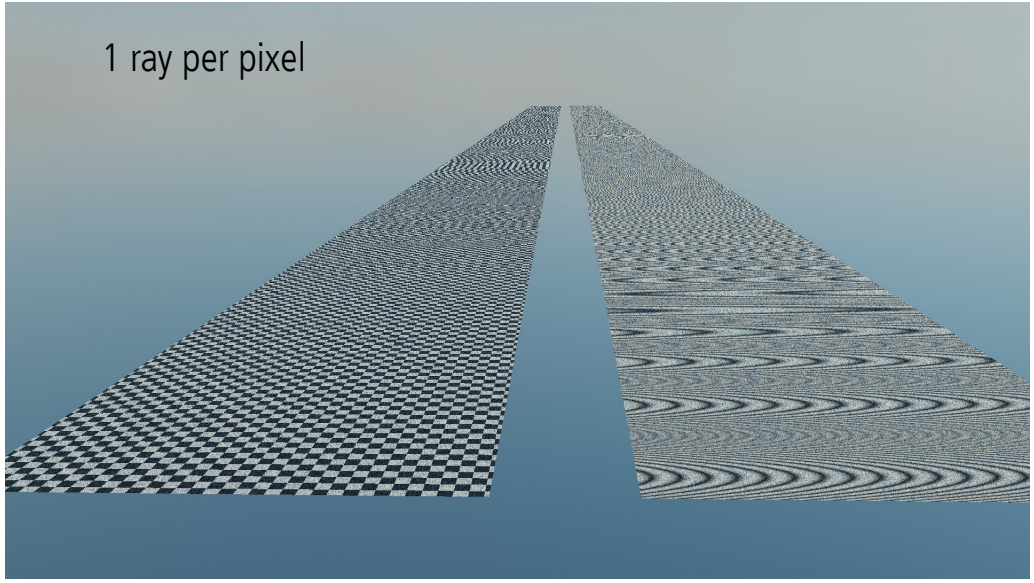
Generating camera rays: multiple rays

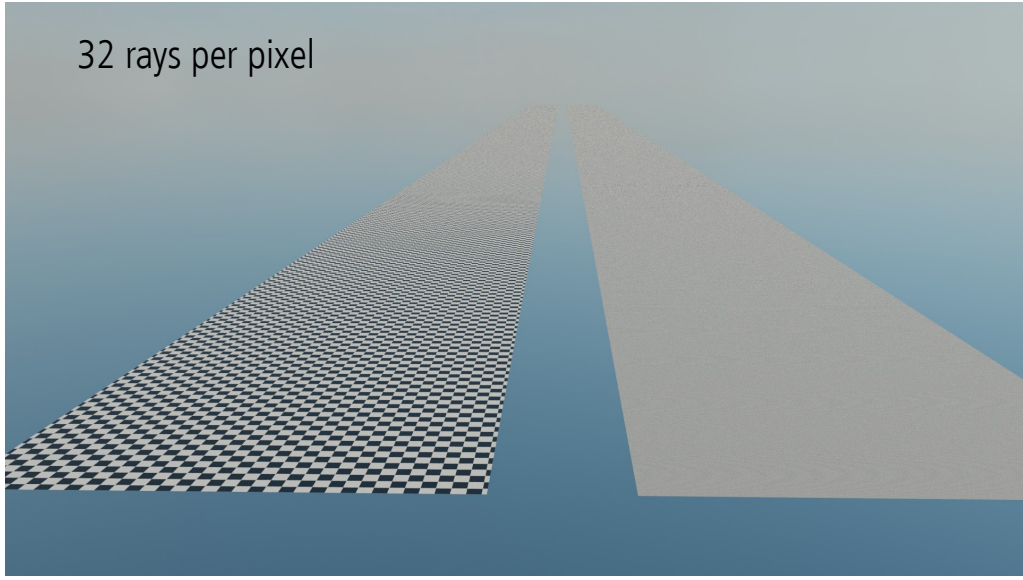
- It is possible, and in practice always desired, to generate multiple rays for each pixel.
- It is important to note that large portion of the scene is actually represented by only one pixel.
- Since pixel can represent only one color, it is important to use multiple rays to obtain the color which is the most representative for that part of the 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

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;
    }
}
...
```

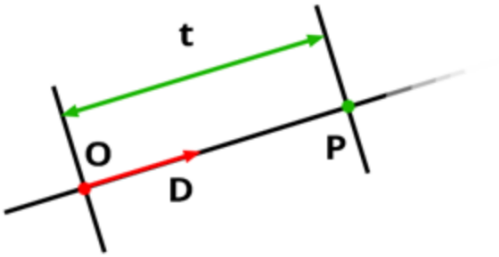
- Generated camera rays are **traced** into the 3D scene
  - Loop over all 3D objects in scene and test them for intersection with camera ray → visibility test!
- In this step, we are interested in a shape of 3D object.
- Once intersection is determined material will be used during the shading step.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

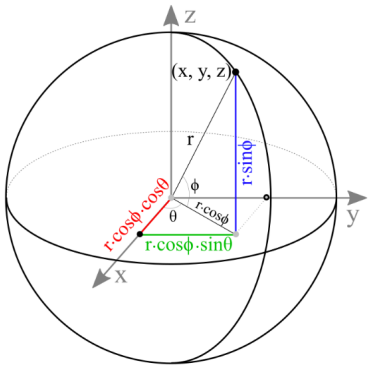
Testing object intersections

- 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:

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

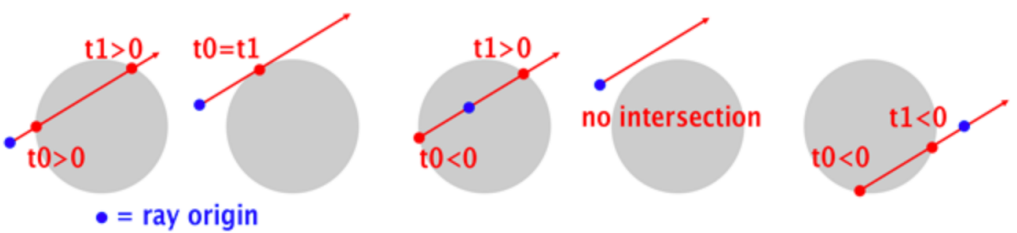
  - $\Delta > 0$: ray intersects sphere in two points (t_0 and t_1):

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

  - $\Delta = 0$: 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

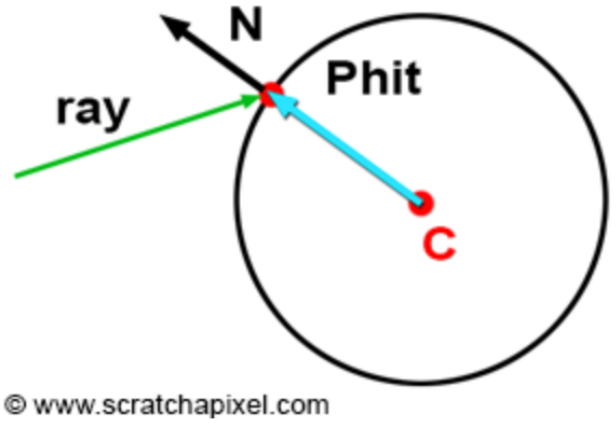
- Now we know t_0 – closest intersection of ray with a sphere
- To find intersection point, we use parametric ray equation:
  - `P(t) = O + t * D`
- Finally,
  - `vector3 Phit = O + D * t`

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

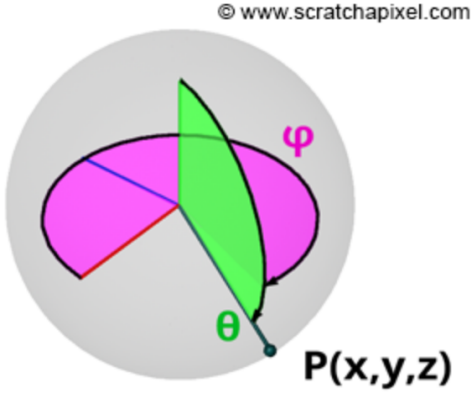
- Normal calculation in intersection point depends on shape representation
- For implicit sphere:
  - `vector3 N = normalize(Phit - C)`



Ray-sphere intersection: texture coordinates

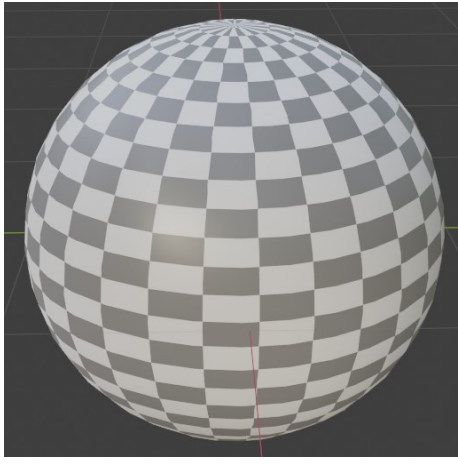
- Calculation of texture coordinates depends on shape representation
  - For parametric and implicit surfaces it can be calculated on the fly
  - For polygonal (e.g., triangle) mesh, texture coordinates are stored per vertex. Intersection point is used to interpolate texture coordinates per triangle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

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



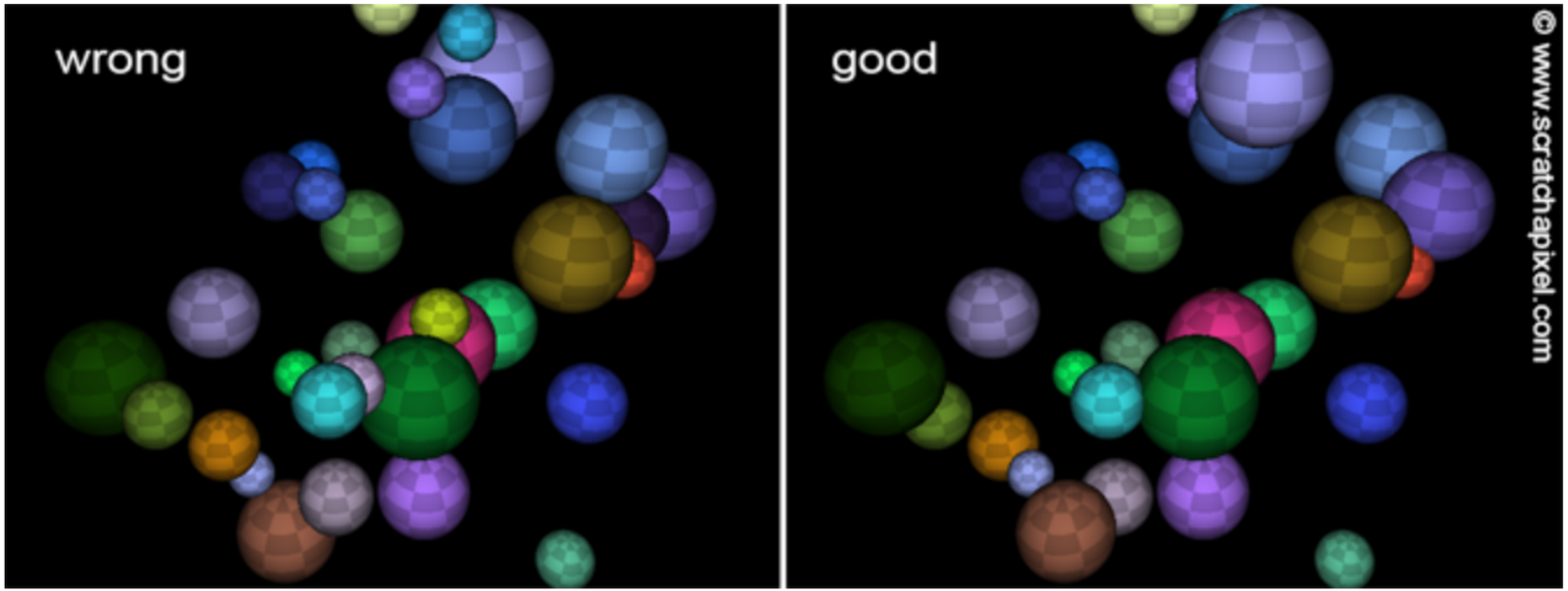
- Texture coordinates for sphere are simply spherical coordinate

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



Ray-sphere intersection: intersection order

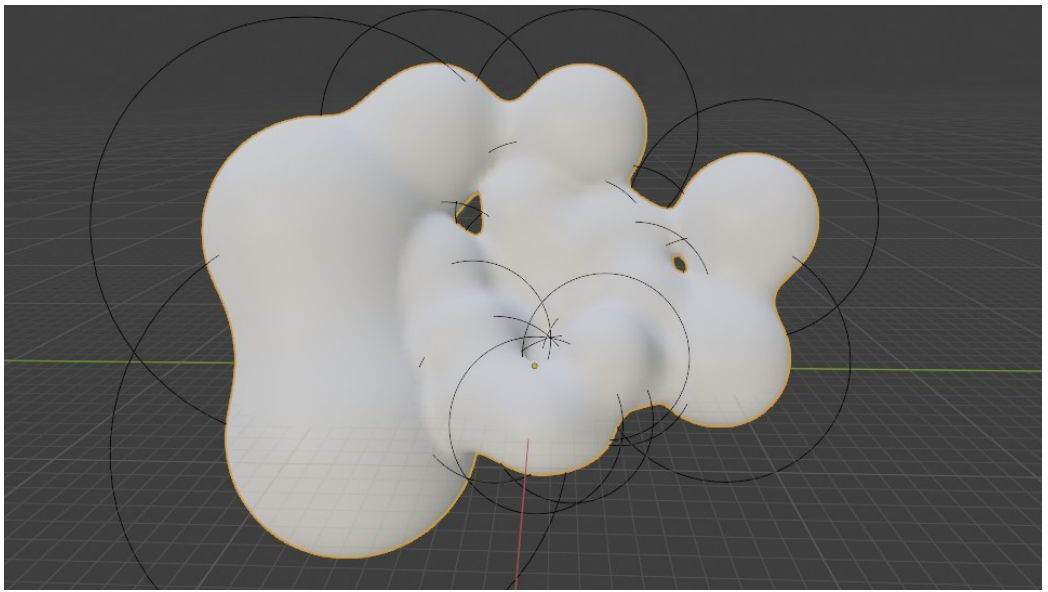
- If scene contains multiple objects then certain depth sorting is needed
- The solution is to keep track of closest intersection distance → closest t

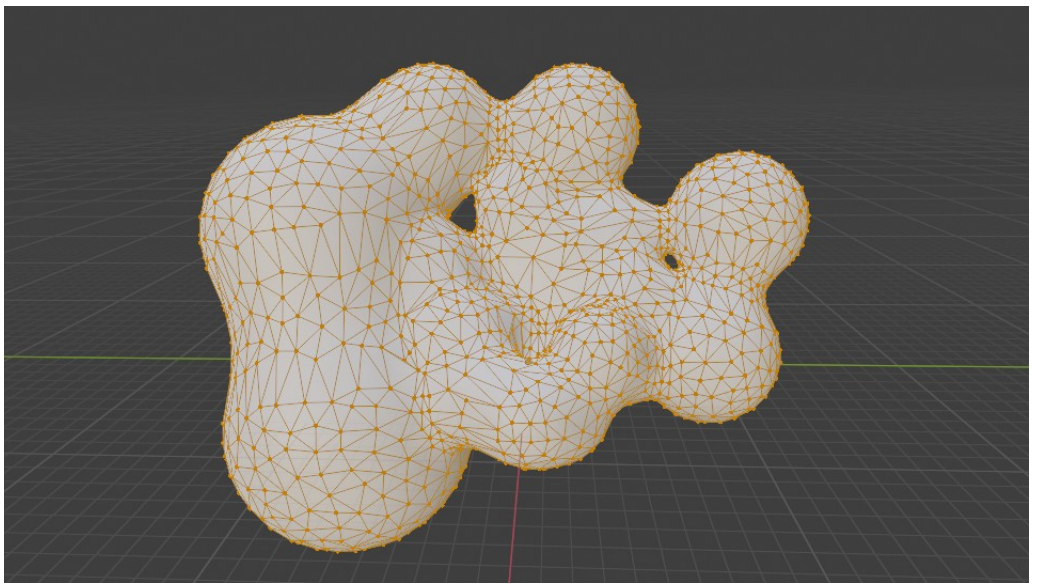


Intersecting other shapes

- 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] = shadingResult;
        }
        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

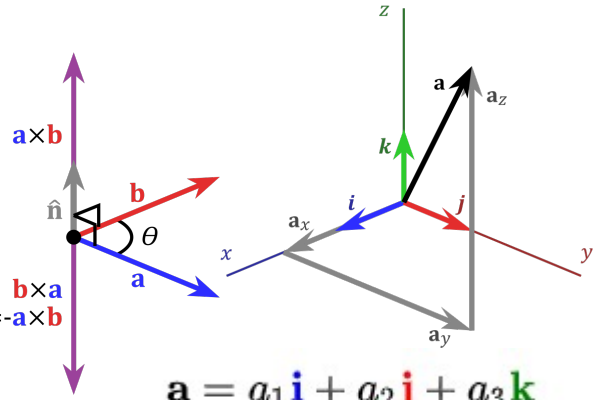
- Basic ray-triangle intersection is straight forward, complexity comes due to multitude of different cases which must be accounted for
  - Thus, there are several algorithms that have been developed and are being developed
- Main questions:
  - Does ray intersect a plane defined by a triangle?
  - Does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

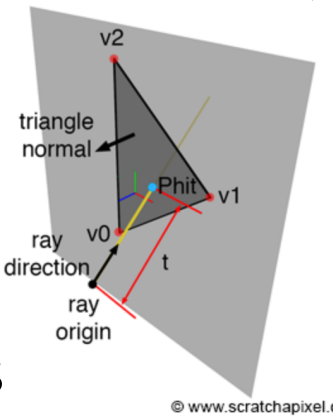
Intersecting triangle primitive: tools

- 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!**





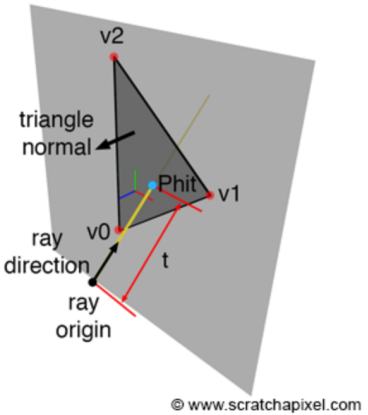
$$\mathbf{a} = a_1\mathbf{i} + a_2\mathbf{j} + a_3\mathbf{k}$$
$$\mathbf{b} = b_1\mathbf{i} + b_2\mathbf{j} + b_3\mathbf{k}$$
$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$
$$\begin{bmatrix} s_1 \\ s_2 \\ s_3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

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

Intersecting triangle primitive: intersecting plane

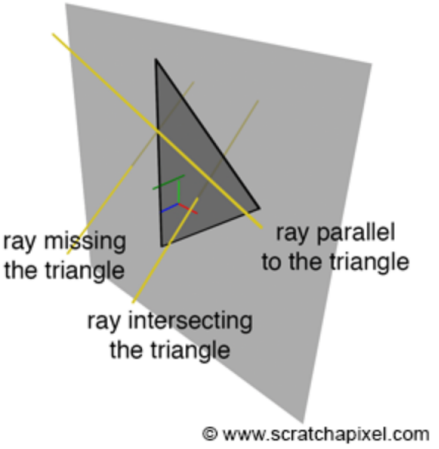
- Intersected point in somewhere on ray:
  - `Phit = P(t) = O + t * R`
- Plane equation :
  - `Ax + By + Cz + D = 0`
  - `D = -(Ax + By + Cz)`
  - `A, B, C` are coordinates of plane normal `N = (A,B,C)`
    - `N` is calculated using triangle vertices
  - `D` is distance from origin `(0,0,0)` to the plane
    - 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 :

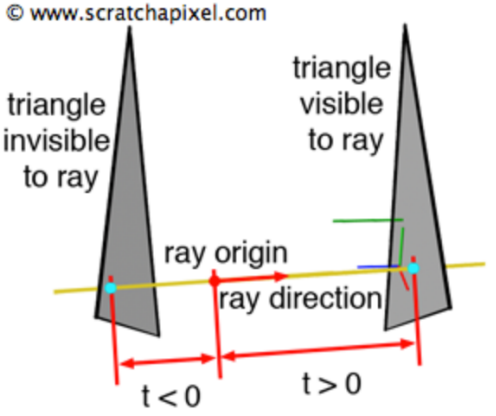
$$t = -\frac{N(A,B,C) \cdot O + D}{N(A,B,C) \cdot R}$$



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

Testing intersections: trace function

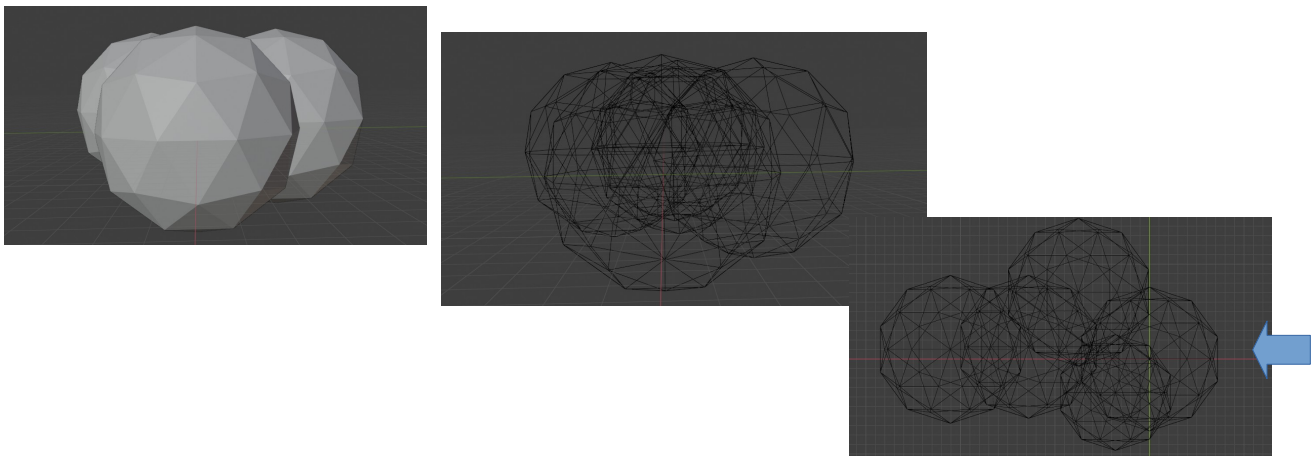
```
bool trace (Ray ray, Object objects, &objIdx, &triIdx)
{
    bool intersect = true;
    for (int k = 0; k < objects.nObjects; ++k)
    {
        for (int n = 0; n < objects[k].nTriangles; ++n)
        {
            if(rayTriangleIntersect(ray, objects[k].triangle[n])
            {
              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

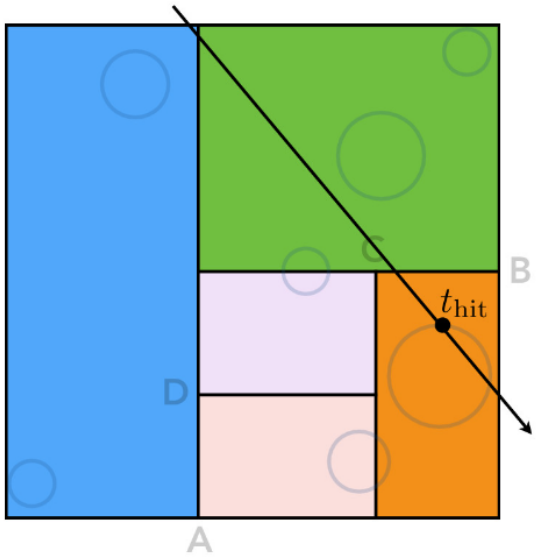
- Trace function returns `objIdx, triIdx` and `tNearest`
  - This is needed to calculate intersection context used in shading:
    - **intersection point**,
    - **normal**,
    - **texture coordinates**, etc.
- Trace function can also directly calculate intersection context – depending on rendering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

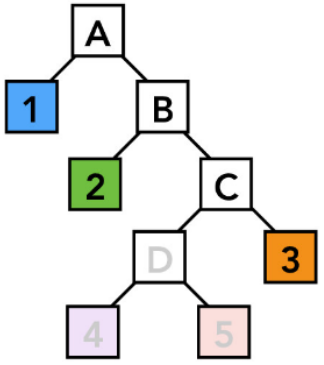
Testing intersections

- 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.



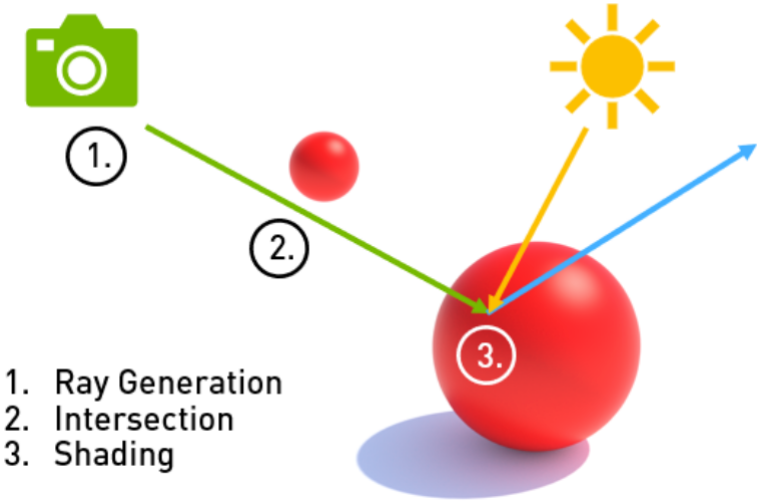


Intersection found

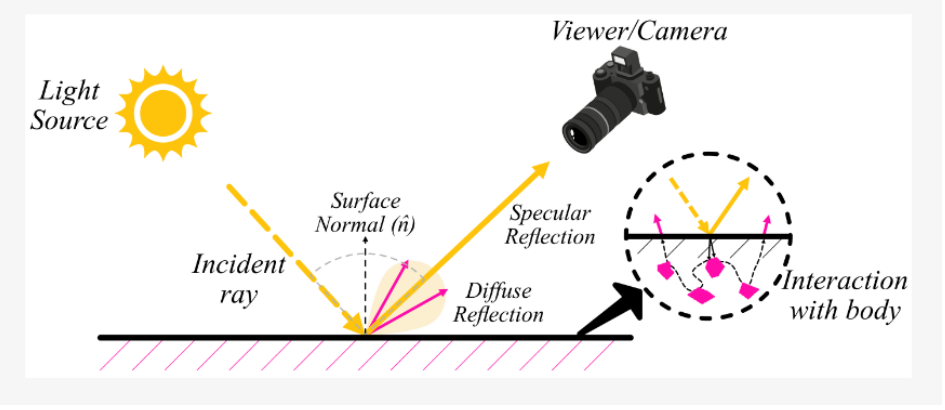
https://cs184.eecs.berkeley.edu/sp22/lecture/10/ray-tracing-acceleration

Shading

- Once **intersection of camera ray with object** is found, we need to calculate color and intensity of that point → **shading**.
- **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

Shading and light transport

- Shading very much depends on incoming light on the surface
- "Gathering" incoming light on shading point → **light transport**.
  - In real world: light is emitted from light source, bounces around the scene and small portion enters camera
  - Ray-tracing: trace rays from camera to 3D scene objects → **backward tracing**
- Light is gathered by tracing rays from intersection point to the 3D scene.
  - Those rays are called **secondary rays:**
    - Ray casted toward light is called **shadow ray**
    - Ray casted from mirror is called **reflection ray**
- **Secondary rays used in light transport are solving visibility problem!** For example secondary ray casted towards light connects two points: intersection point and light position. And it is important to evaluate if these two points are visible to each other – if there is another object on this segment.

Light transport

- Much of the shading process in shading point relies on simulating light bouncing off of the objects or light traveling through objects in 3D scene → **light transport**
  - Essentially, seeing an object means that light is reflected by this object in the direction of camera/eye.
  - Some (a lot actually!) objects in a 3D scene may reflect light in direction where camera is not placed, but due to indirect reflection, the light is reflected by another object in the scene*.
  - Opaque surfaces (e.g., shiny) cause light reflection
  - Transparent surfaces cause light refraction while entering, traveling and exiting such objects
- Light transport must simulate light scenarios in acceptable amount of time and yet providing best visual accuracy possible.
  - Light transport algorithm is strategy to solve light transport. Many strategies for light transport have been developed over time (and still are!)
- Simulation of light bouncing in the scene requires:
  - Models which determine how ray is reflected once it falls on object – for this purposes we use material and its component: BRDF/BSDF
  - Efficient intersection tests for any object in 3D scene computing visibility. Besides computing intersections with first visible objects (objects visible from camera), we also need to compute visibility for arbitrary points in the scene after**. Note that next to ray-tracing we might employ other methods for visibility calculation and thus light transport.

* Note here once more the importance of material information. We said that material information is used during shading process which is true. But material information is also used during light transport to determine how will light bounce. Note that specifically scattering function in material description will give this information. By now we have mostly looked at scattering function as a function that gives us information how much light from camera ray intersection (shading point) will reflect into eye from incoming light direction. But in light transport, the same function is used to determine where the light will scatter given the input direction – thus the name "bidirectional" in BRDF/BSDF.

** This is one of the largest differences between ray-tracing-based and rasterization-based rendering. Rasterization fast to calculate what is visible from camera, but is is not suitable at all to give information which surfaces are visible to surfaces visible from camera and not to mention arbitrary suface visibility which is important for quality light transport. There are some approaches in radiosity light transport utilizing hemisphere and hemicube but they are extremly slow – INVESTIGATE!

Note: light transport and visibility

- Important ingredient for photo-realistic images is high-quality light transport algorithm.
- Light transport, as discussed relies on visibility calculation.
- By now, we have introduced ray-tracing as one possible solution for visibility calculation. Another one is rasterization.
- When considering rendering algorithm/software, it is important to understand which kind of light transport it employs and which visibility calculation method it uses.
  - Visibility method dictates light transport and thus realism of rendered images.
  - Amount of global illumination effects that can be simulation depends on light transport algorithm.
- For example, light transport algorithms employing ray-tracing as visibility method are path-tracing and bidirectional path-tracing.
- Practical tip: In DCC Tools, users are often provided with faster and lower quality rendering methods and slower and higher quality rendering methods. First one is used for preview and modeling. Second one are used for final renders.

<IMAGE: Workbench vs EEVEE vs Cycles in Blender>

Light transport based on ray-tracing

- Reminder: difference between visibility method and light transport algorithm: light transport algorithm employs visibility for calculation.
- Classical example of light-transport algorithm utilizing ray-tracing for computing visibility is **Whitted-style-ray-tracing.** This method employs **backward tracing**.
- Whitted light transport algorithm simulates*:
  - Diffuse and specular (glossy?) surfaces
  - Reflections between objects. <Example: mirror-like and diffuse (diffuse one will be reflected in mirror-like surface) and two mirror-like objects (both objects will be reflected into each other)>
  - Refraction when light enters and exits transparent objects. <Example: transparent surface and bending rays due to refraction while light passing through them>

<IMAGE: reflections and refractions vs mirror like surfaces without reflections>

- When we discussed materials, we introduced concepts of reflection and refraction as well as their physical foundations. These are used in Whitted light transport for calculating light bouncing in 3D scene which contains mirror-like and transparent surfaces:
  - Reflection direction depends on surface orientation (normal - property of object shape) and incoming light direction.
  - Refraction direction depends on surface normal (property of object shape), incoming light direction and index of refraction (property of material).
    - Tip: Higher refractive index of transparent object → stronger specular reflections.
    - Tip: Higher angle between normal and angle of incidence → more light is reflected (Fresnel effect)

* Before Whitted introducted Whitted-ray-tracing light transport, there were already reflection models capable of simulating diffuse and specular surfaces – Phong scattering model. But note that specular surfaces (as they are mirror like) should reflect other objects in the scene – not only appear mirror-like. Also, nect to specular reflection we can also observe specular refraction that happens for example on glass. For exactly this effects, we need light-transport algorithm coupled with scattering model to produce them.

Whitted rendering method overview

Refraction or reflection rays falling on surface may have following cases*:

- If surface is opaque and diffuse, then use diffuse scattering model (e.g., Phong) and other material properties (e.g., color) to calculate reflected color and intensity. As discussed, we need to know information about incoming light (and if it obstructed from those - shadow) and thus simple raycasting from intersection point to light is performed.
- If surface is opaque and mirror-like, use specular reflection model to determine new ray from intersection
- If surface is transparent, use specular reflection and refraction model to determine reflected and refracted ray from intersection point.

<IMAGE: different reflection/refraction possibilities: mirror-mirror-diffuse and how second first mirror take the color of diffuse object.>

<IMAGE: different reflection/refraction possibilities: transparent-transparent-diffuse and how second first transparent take the color of diffuse object. Higher refractive index: stronger distortion (e.g., glass vs diamond)>

* All of those effects are relying on ray-object intersections and visibility calculation performed by ray-tracing!

Whitted light-transport properties: recursive nature

- Shading process, which starts when camera ray intersection with object in the 3D scene is found, requires gathering light at shading point using light transport. Direct illumination only is about computing visibility between intersection point and light source. Indirect (global) illumination is much more complex:
  - Each visible intersection of a ray with a surface produces more rays in reflection/refraction direction.
  - If scene contains lot of reflective (mirror-like) objects, then the shading point is reflective and another ray is generated and traced into scene. This ray can intersect another object, if this object is reflective, then another ray is generated and traced into scene → as we can see, this process can continue to infinity and it is of **recursive nature**.
  - More complex case is if 3D scene also contains transparent objects. In this case, each intersection causes two new rays that are traced in the scene. Note that in this case we have two paths that have to be processed.
  - To stop generation of new rays for path, it is required to set maximum number of bounces (recursions).
  - Based on the discussion above, we can conclude that render time increases exponentially with the level of depth

    <IMAGE: tracing rays and recursion>

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

<IMAGE: tree of rays>

Shading and light transport: algorithm

- TODO

Notes on ray-tracing based rendering

- Pros:
  - 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.
- Cons:
  - 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.

* 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).
** 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

Code

Code

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

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, they are mostly written for CPUs
- Porting ray-tracers to GPU was done in 90s
- Ray-tracing ideas are often used on GPU and rasterizer renderers, but not to a full extent 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.