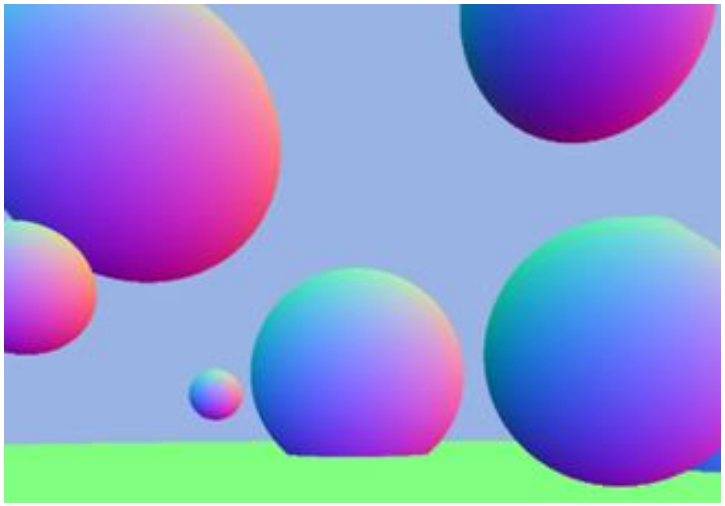
Raytracing
I – Raycasting

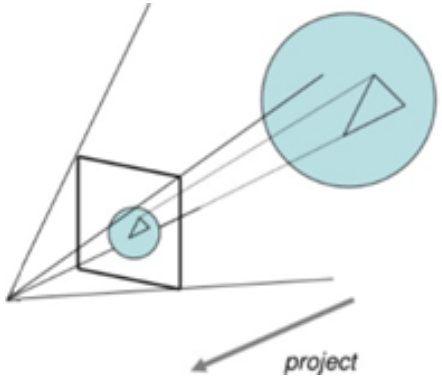


Stefan BORNHOFEN

Rendering Algorithms

Rasterization

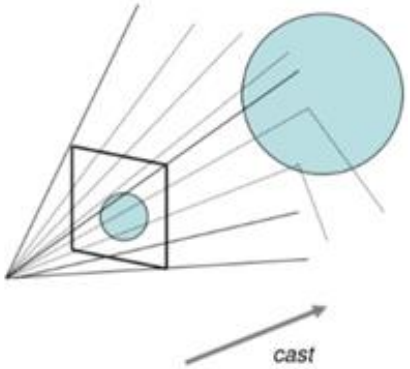
- Efficiency
- Photorealism



- Project polygons onto picture plane
- Efficient hardware. OpenGL, DirectX

Ray Tracing

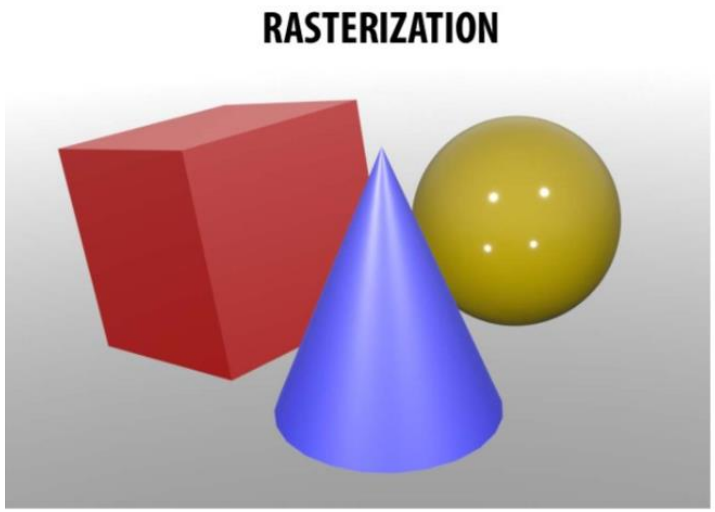
- Photorealism
- Efficiency



- Cast light rays into scene through picture plane.

Rasterization is based on a pipeline model in which primitives are rendered independently. However many visual effects are caused by interactions between the objects. It is hard (but not impossible) to simulate these effects with rasterization techniques.

Rendering Algorithms



In addition to direct illumination, raytracing allows incorporating more complex light interactions, such as:

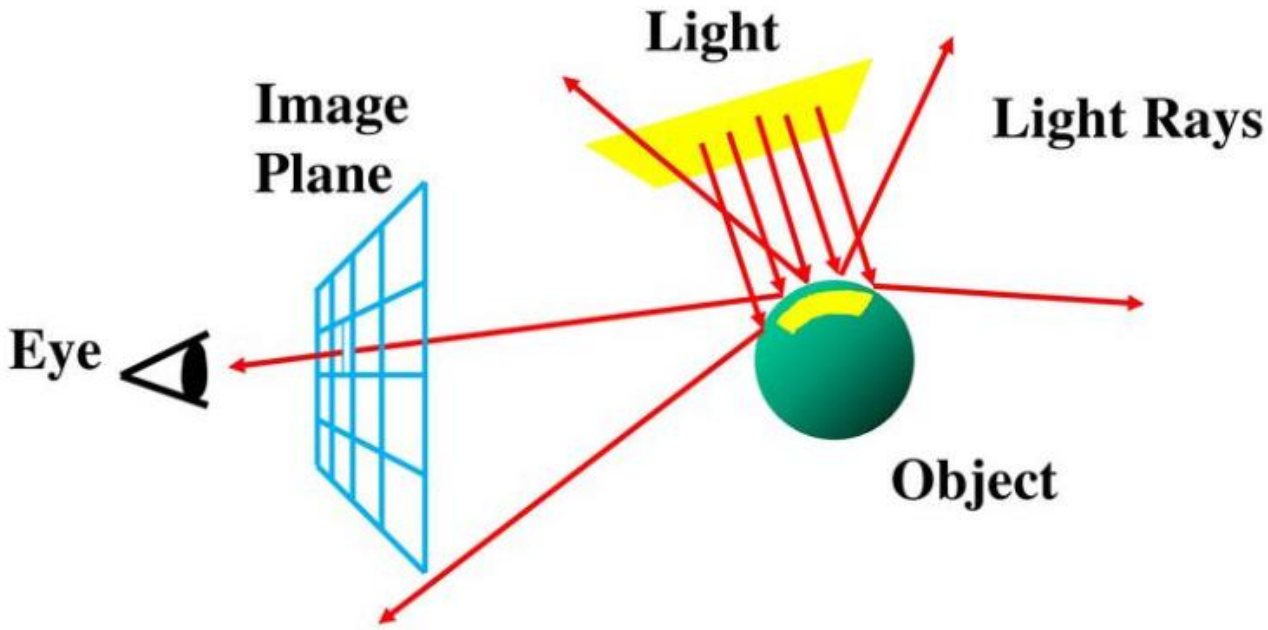
- Shadows
- Reflexions (mirrors)
- Refractions (transparent objects)

Still, many real-world light phenomena cannot be taken into account.

"Rasterization is fast, but needs cleverness to support complex visual effects. Ray tracing supports complex visual effects, but needs cleverness to be fast." (David Luebke, Nvidia)

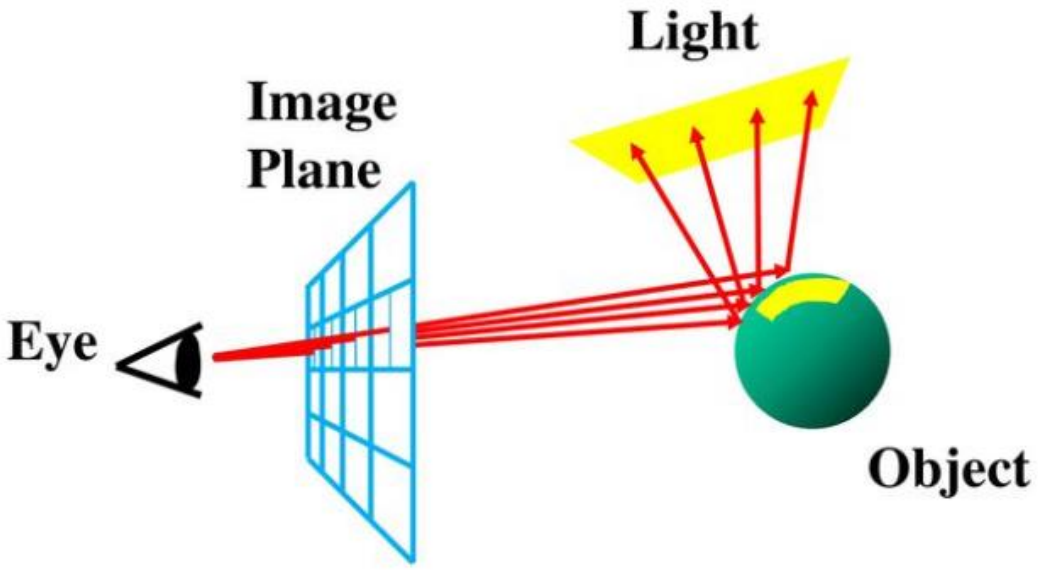
Forward Raytracing

- Also known as light ray tracing and photon tracing
- Follows the light particles (photons) from the light source to the object
- Scattering produces many additional rays
- Most rays will go to waste because they never contribute to the final image



Backward Raytracing

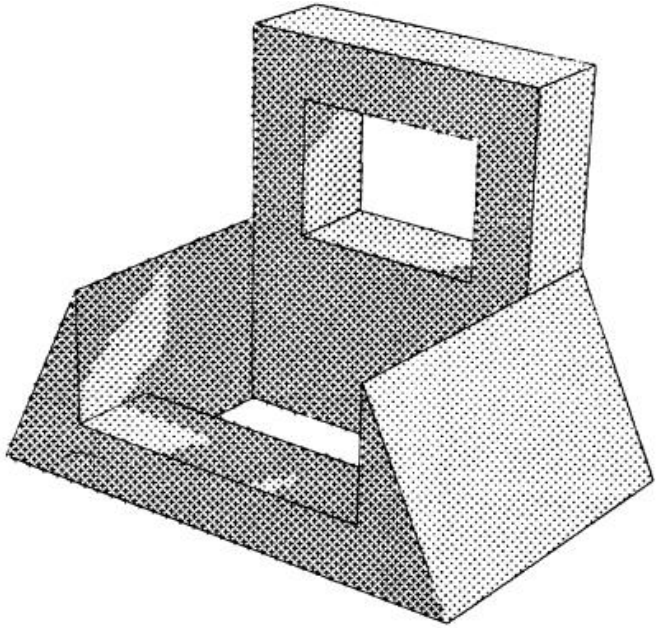
- A photon's path obeys time-symmetry
- Only rays that reach the eye matter
- Reverse direction and cast rays
- At least one ray per pixel



History

1968, Arthur Appel
Some techniques for shading machine renderings of solids

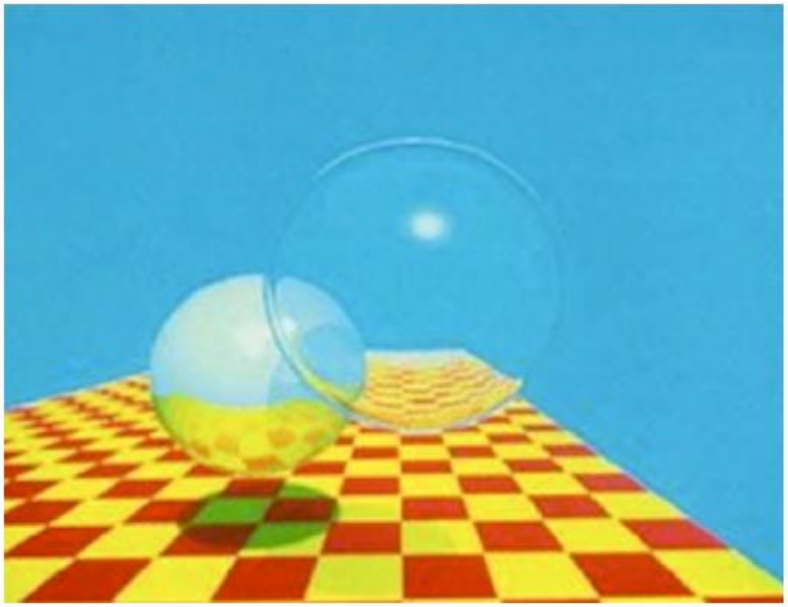
- First ray casting algorithm
- Ray casting renders the scene by emitting a ray from the observation point through each pixel and finding the closest object that blocks the light path.
- Only two rays were involved:
  - Eye rays emitted by the eye to find the intersection point
  - Shadow rays sent from the intersection point to the light, to see if the point is in the shadow



History

1980, Turner Whitted
An improved illumination model for shaded display

- First global illumination model: illumination is influenced by light sources and other objects in the scene
- Primary rays originating directly at the camera
- Secondary rays to account for indirect effects like shadows, reflection, and refractive transmission
- Recursively evaluating the light being transported along these rays



T. Whitted, Communications of ACM, 1980

Render time: 74 min ☺

History

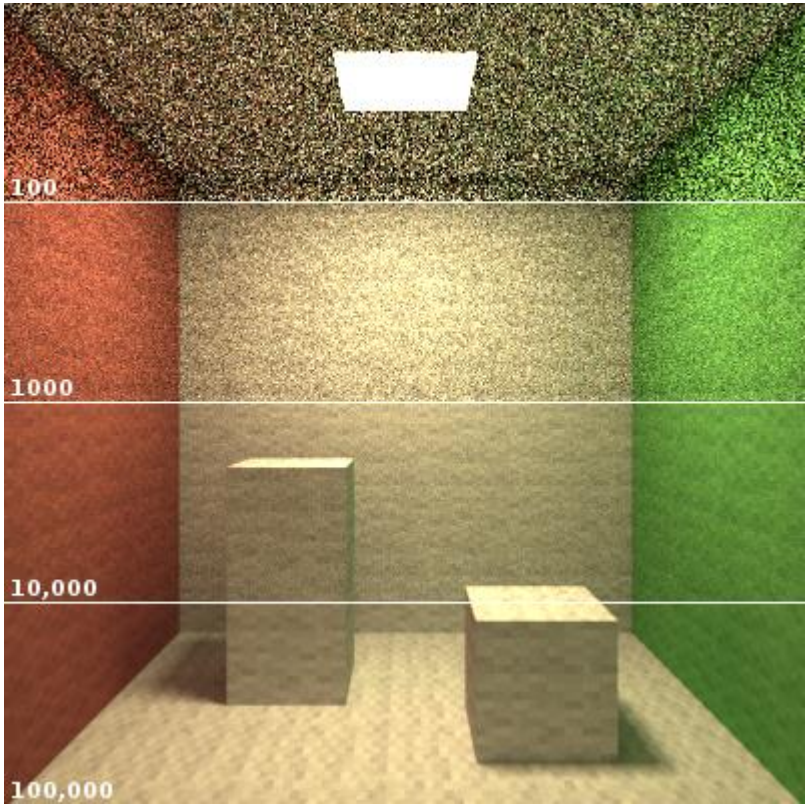
1984, Robert Cook
Distributed ray tracing

- Today: **Distribution** ray tracing
- Allows for soft phenomena by averaging multiple rays distributed over an interval.
- Sampling captures effects such as
  - depth of field
  - glossy reflections
  - soft shadows
  - motion blur
- In order to achieve a sufficient quality without noise, distribution ray tracing requires to generate a relatively large number of samples, and is usually quite costly.

History

1986, James Kajiya
The rendering equation

- Diffuse inter-reflections
- Conceptual basis for all methods of global illumination: Bidirectional reflectance distribution function (BRDF) which describes how light scatters from surfaces.
- Secondary rays have to be emitted from all surfaces
- Monte Carlo path tracing



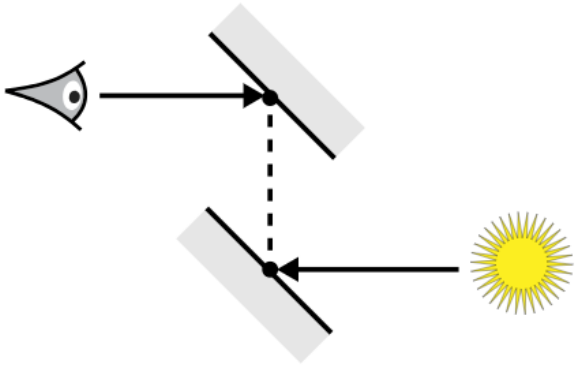
Monte Carlo path tracing uses random sampling to incrementally compute a final image. The approach causes noise, which dissipates by letting the algorithm generate more samples.

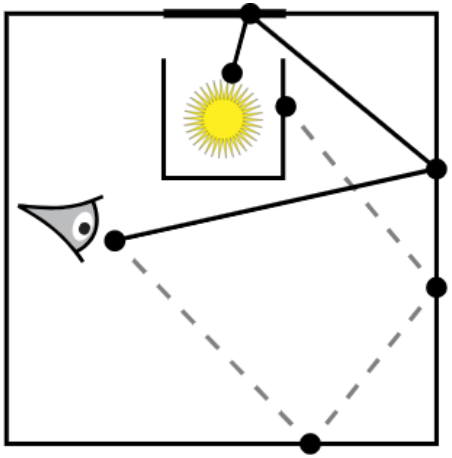
History

1993, Eric Lafortune

Bidirectional Path Tracing

- Difficult lighting settings can be handled more effectively by constructing paths that start from the camera on one end, and paths that start from the light on the other end.
- Connect both paths in the middle with a visibility ray.





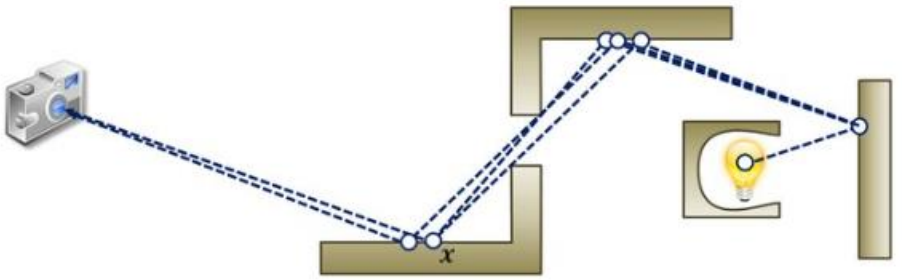
(a) Bidirectional path tracing with 25 samples per pixel

(b) Standard path tracing with 56 samples per pixel (the same computation time as (a))

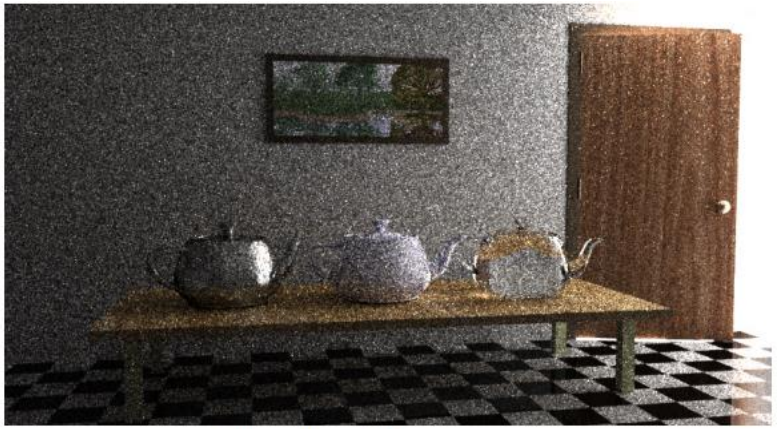
History

1997, Eric Veach

Metropolis Light Transport



- Generates a sequence of paths.
- Each path is found by mutating the previous path.
- When a path that makes a large contribution to the image is found, explore similar paths by applying small perturbations to it.
- Efficient in difficult settings where most of the light transport happens along a small fraction of all of the possible paths through the scene.



(a) Bidirectional path tracing with 40 samples per pixel.

(b) Metropolis light transport with an average of 250 mutations per pixel [the same computation time as (a)].

Today

- Raytracing has become a standard in the marketing, film and video games industries
- The latest GPU capacities allow for real-time performances (Nvidia RTX)
- Current research mostly revolves around acceleration techniques of existing raytracing methods

IKEA catalogue

Cars

Cyberpunk 2077

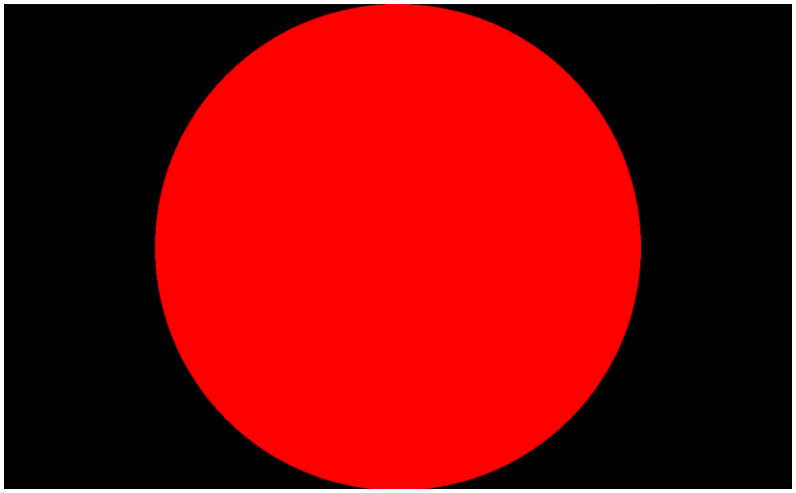
Let’s write a Raytracer!

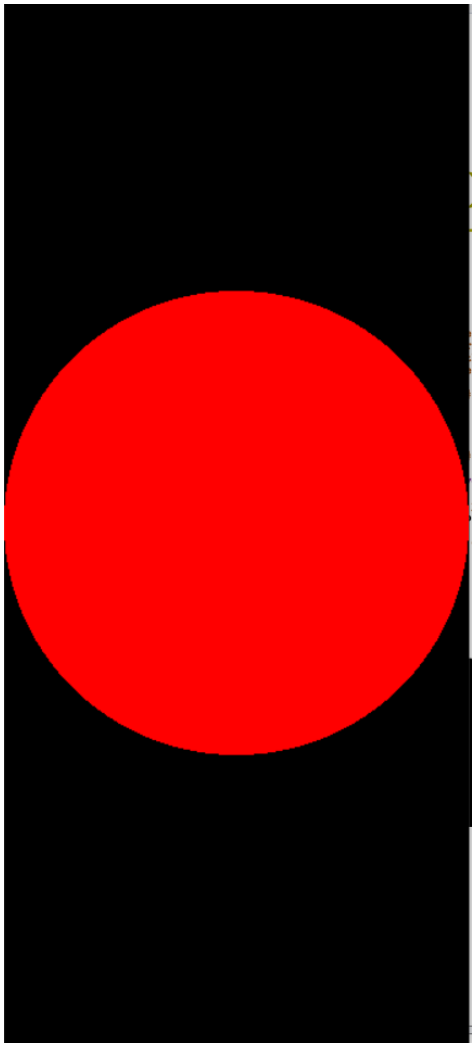
CPU or GPU?

CPU	GPU
Free choice of your favorite programming language	Shader or GPU programming languages (GLSL, CUDA C, OpenCL) with reduced capacities: no recursion, no dynamic memory allocation, limited local memory, etc.
All data in RAM	Host-device data transfer
Slow	Fast
Image saved to file	Image ready to render
Good for complex high quality raytracing	Good for simplified real-time raytracing

Your work

- Set up a GLUT window
- Draw a quad that will serve as our image plane
- Write basic vertex and fragment shaders
  - The vertex shader has no work
  - The fragment shader computes UV coordinates, where the origin is in the center and the shorter side has length 1. Draw a disk of radius 1 to validate the coordinates.
- The disk must always stay in the center and touch the borders of the screen





Quad init and draw

```
// init quad
glGenVertexArrays(1, &vaoquad);
glBindVertexArray(vaoquad);
const GLfloat vertices[] =
  { -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, 1.0f, 1.0f, -1.0f };
glGenBuffers(1, &vboquad);
glBindBuffer(GL_ARRAY_BUFFER, vboquad);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);
```

```
// draw quad (in the GLUT display function)
glViewport(0, 0, SCREEN_X, SCREEN_Y);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glBindVertexArray(vaoquad);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
glutSwapBuffers();
```

Shader setup

Vertex Shader:

```
in vec4 in_position; // [-1,1] x [-1,1] quad
void main()
{
    gl_Position = in_position; // no work
}
```

Fragment Shader:

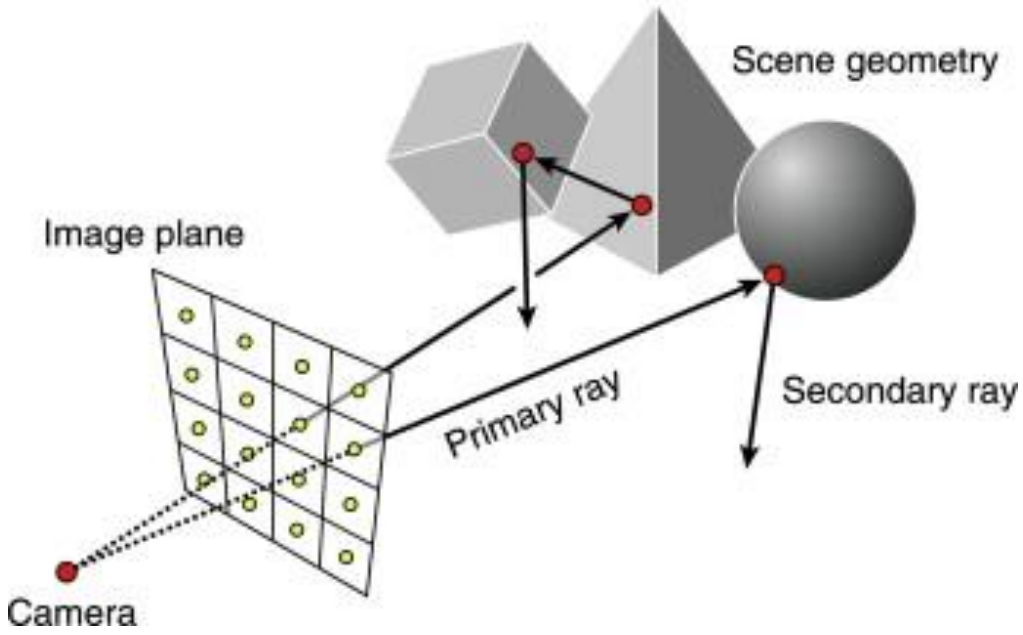
```
uniform float view[2]; // send SCREEN_X and SCREEN_Y
vec2 UV; // the UV coordinates of this pixel on the canvas
out vec4 fColor; // final color

vec2 computeUV() // gl_FragCoord in [0,SCREEN_X] x [0,SCREEN_Y] => UV
{
  return ... gl_FragCoord ...
}

void main(){
  UV = computeUV();
  vec3 color = vec3(0.f, 0.f, 0.f);
  if (length(UV)<1) color = vec3(1.f,0.f,0.f);
  fColor = vec4(color,1.0f);
}
```

Primary and secondary rays

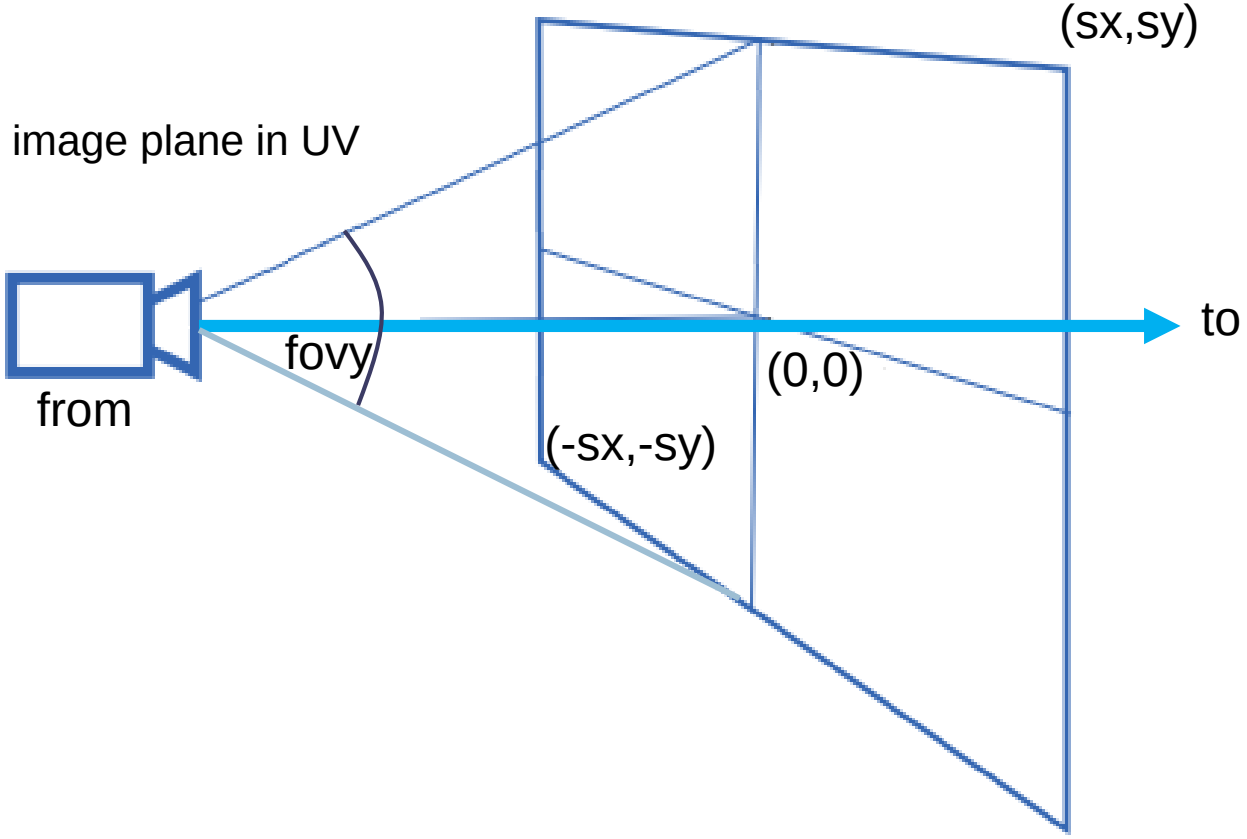
The rendering process requires generating a ray for each pixel, casting this ray into the scene, and looking for intersections between this ray and any surface in the scene. They are called **primary rays** (or camera or eye rays) because they are the first rays cast into the scene. **Secondary rays** are spawned from primary rays at the intersection points.



Primary rays

The view can be configured by

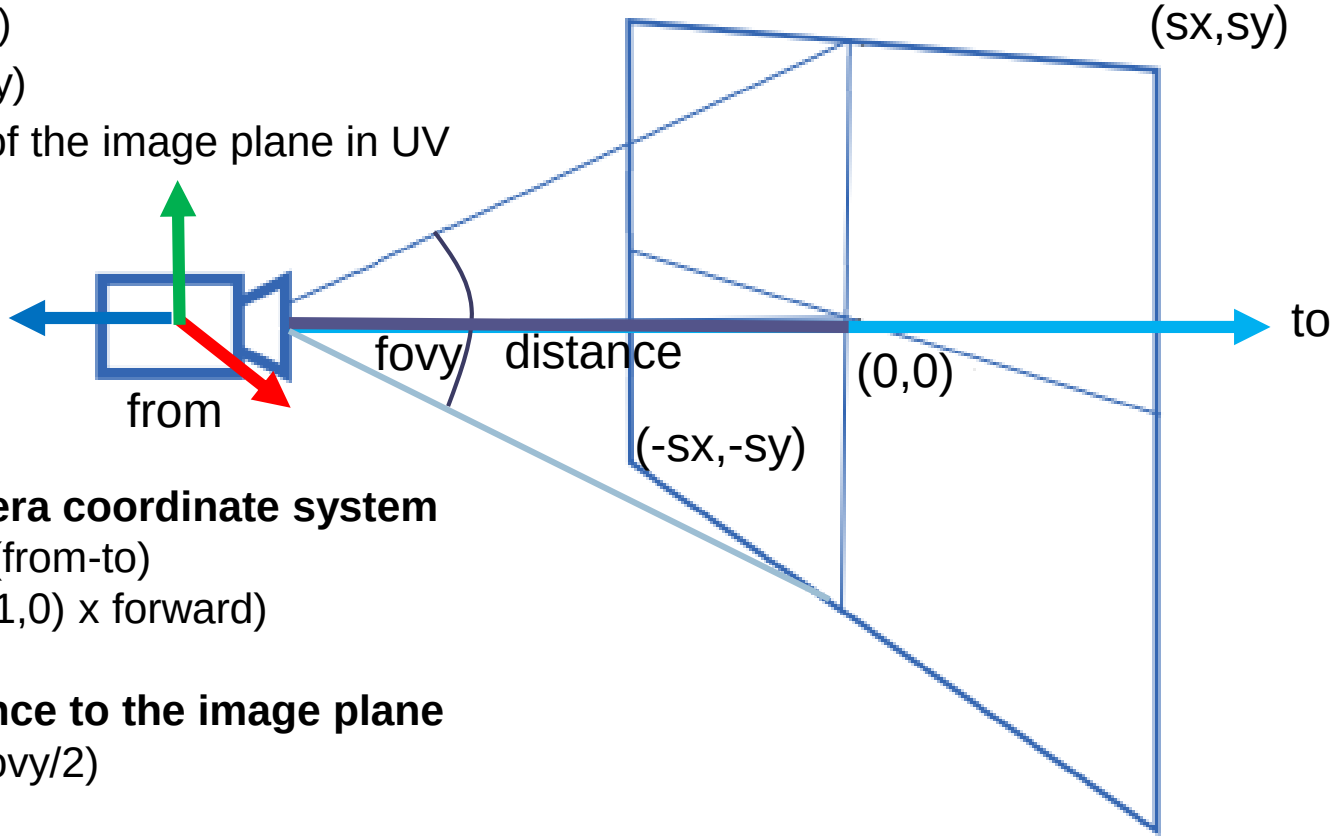
- camera position (from)
- camera target (to)
- Field of view (fovy)
- The size (sx,sy) of the image plane in UV



Primary rays

The view can be configured by

- camera position (from)
- camera target (to)
- Field of view (fovy)
- The size (sx,sy) of the image plane in UV



Construct the camera coordinate system

forward = normalize(from-to)

right = normalize((0,1,0) x forward)

up = forward x right

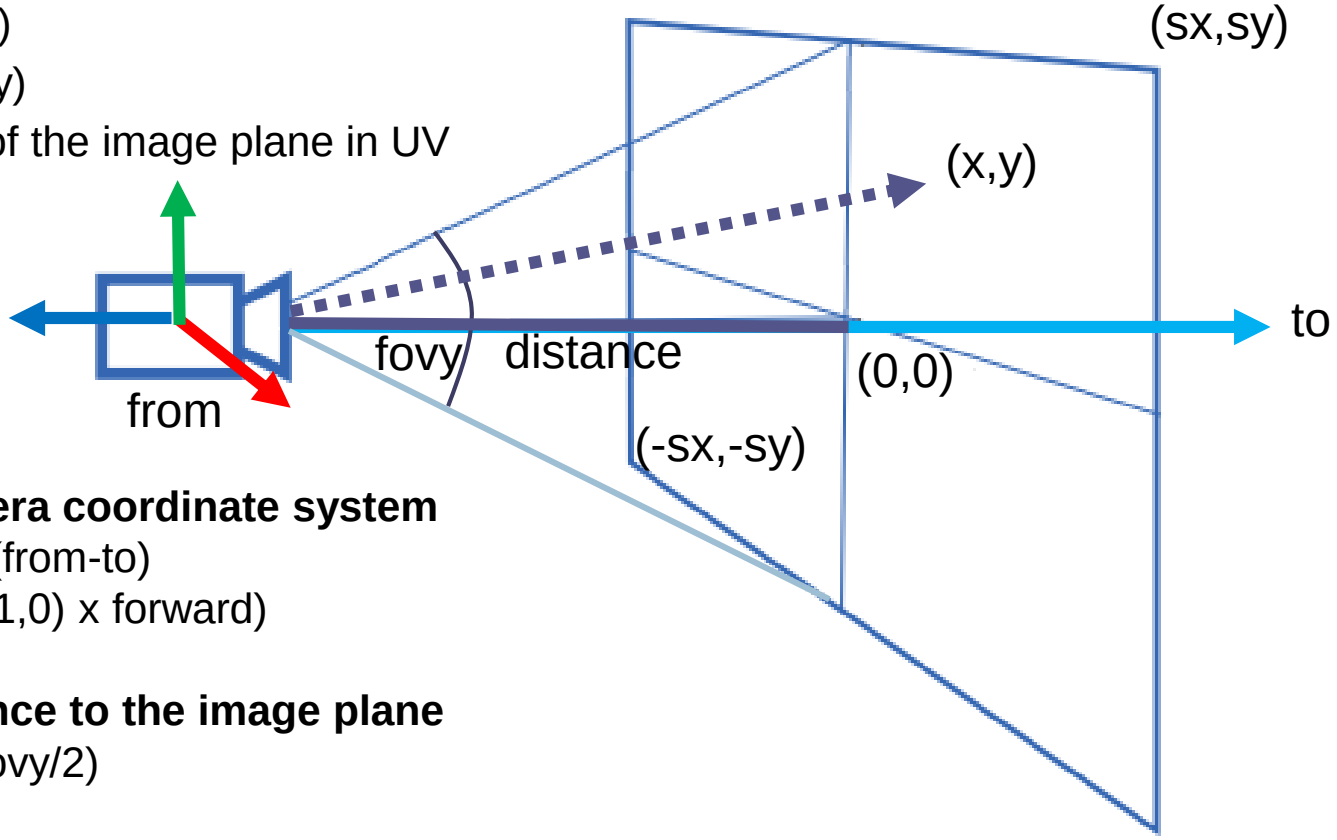
Compute the distance to the image plane

distance = sy / tan(fovy/2)

Primary rays

The view can be configured by

- camera position (from)
- camera target (to)
- Field of view (fovy)
- The size (sx,sy) of the image plane in UV



Construct the camera coordinate system

forward = normalize(from-to)

right = normalize((0,1,0) x forward)

up = forward x right

Compute the distance to the image plane

distance = sy / tan(fovy/2)

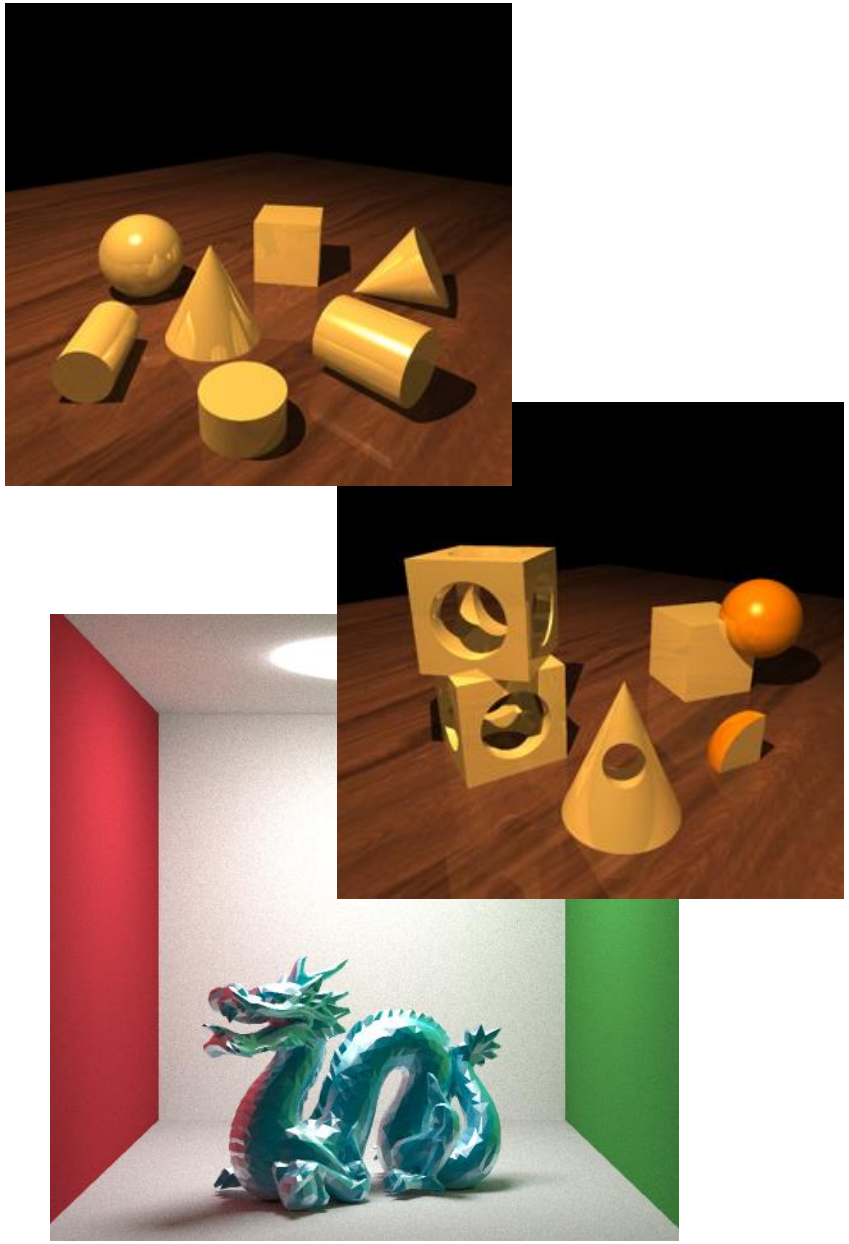
Construct a primary ray going through UV coordinate (x,y)

direction = normalize(x*right + y*up - distance*forward)

ray(t) = from + t*direction

Objects

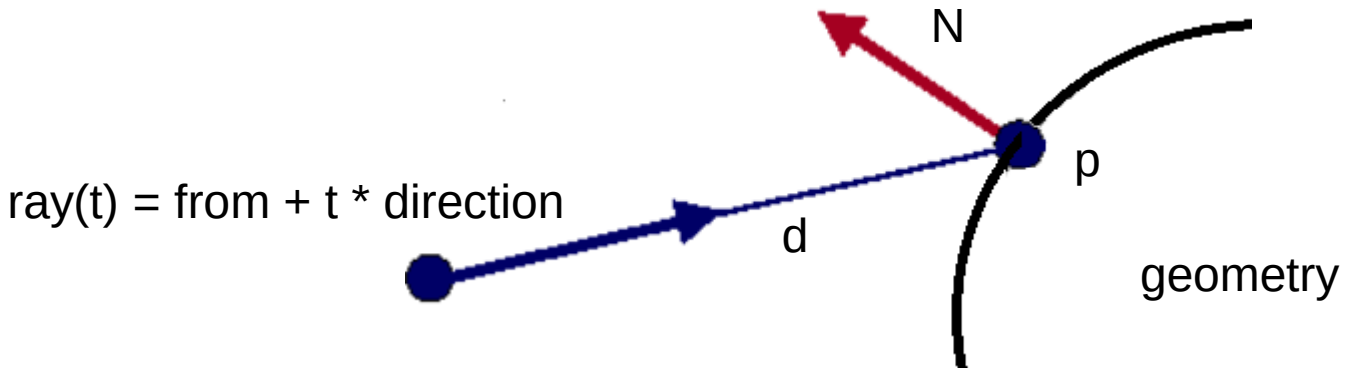
- Simple forms (primitives) often possess an implicit description. Ray intersections can be resolved geometrically or analytically.
- Primitives can be combined together using union, intersection, and difference operations, allowing for more complex models (Constructive Solid Geometry).
- Arbitrary forms can be modeled by a mesh of planar faces. In that case, the raytracing algorithm needs to test ray intersection with each face.



Ray - geometry intersections

We need to compute:

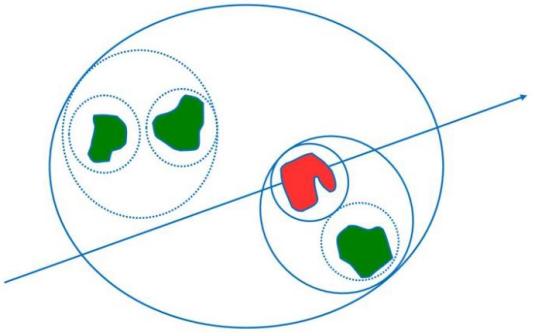
- p: Intersection point
- d: Intersection distance (from camera)
- N: Surface normal at intersection point



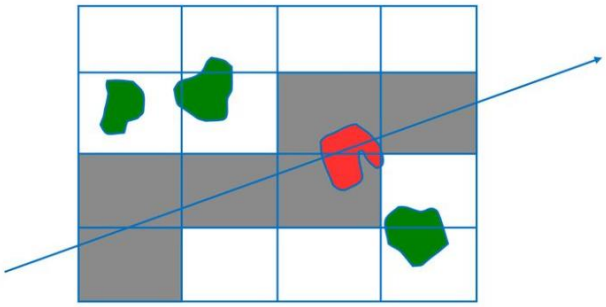
Accelerating Structures

The intersection test is the most expensive task in a ray tracer. The principle of acceleration structures is to help deciding as quickly as possible which objects from the scene a ray is likely to intersect, and reject large group of objects which we know for certain the ray will never hit.

- Bounding volumes



- Spatial partition



In the scope of this course, we do not implement accelerating structures and apply a brute force intersection test.

Ray – Implicit Surface

Many shapes can be represented as implicit surfaces.
Plug the parametric ray equation into the surface equation and solve for t.
We are generally only interested in positive solutions of t.

Ray: $\mathbf{r}(t) = \mathbf{o} + t\,\mathbf{d},\ 0 \le t < \infty$

General implicit surface: $\mathbf{p} : f(\mathbf{p}) = 0$

Substitute ray equation: $f(\mathbf{o} + t\,\mathbf{d}) = 0$

Solve for real, positive roots

$x^2+y^2+z^2=1$

$(R-\sqrt{x^2+y^2})^2+z^2=r^2$

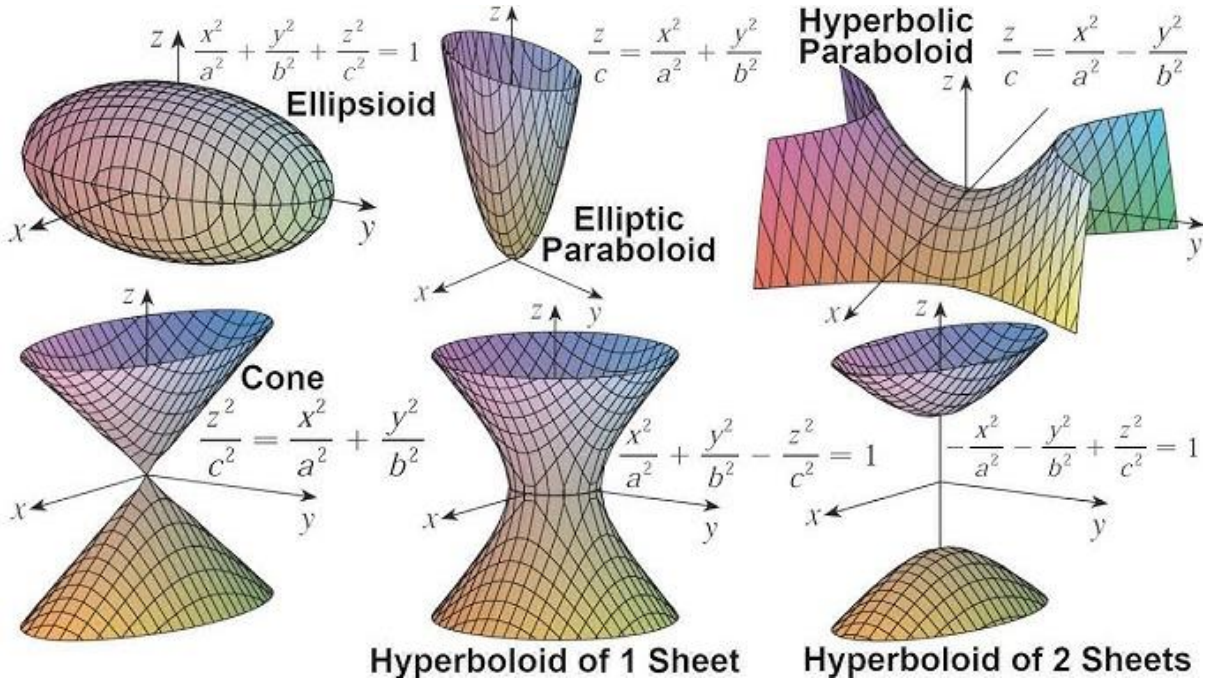
$(x^2+\frac{9y^2}{4}+z^2-1)^3 = x^2z^3+\frac{9y^2z^3}{80}$

Quadric Surfaces

Quadric surfaces are a set of points (x,y,z) satisfying an implicit description in the form of a polynomial equation:

$$Ax^2 + By^2 + Cz^2 + Dxy + Exz + Fyz + Gx + Hy + Iz + J = 0$$

where A, …, J are constants. There are many different families of quadric surfaces such as:

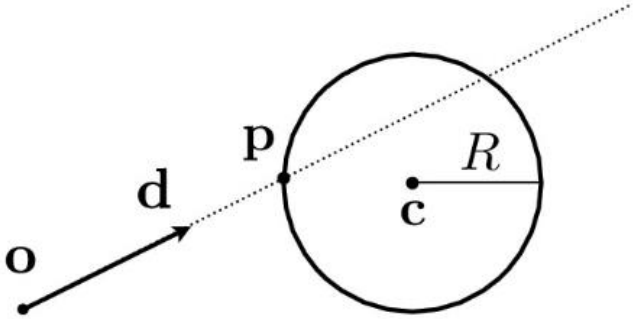


Ray – Sphere

Spheres are firm favorites for ray tracing because they are simple to define mathematically. A sphere is defined by center c and radius R.

Ray: $\mathbf{r}(t) = \mathbf{o} + t\,\mathbf{d},\ 0 \le t < \infty$

Sphere: $\mathbf{p} : (\mathbf{p} - \mathbf{c})^2 - R^2 = 0$



Solve for intersection:

$$(\mathbf{o} + t\,\mathbf{d} - \mathbf{c})^2 - R^2 = 0$$

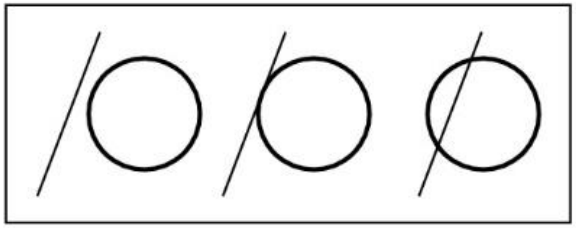
$$a\,t^2 + b\,t + c = 0, \text{ where}$$

$$a = \mathbf{d} \cdot \mathbf{d}$$

$$b = 2(\mathbf{o} - \mathbf{c}) \cdot \mathbf{d}$$

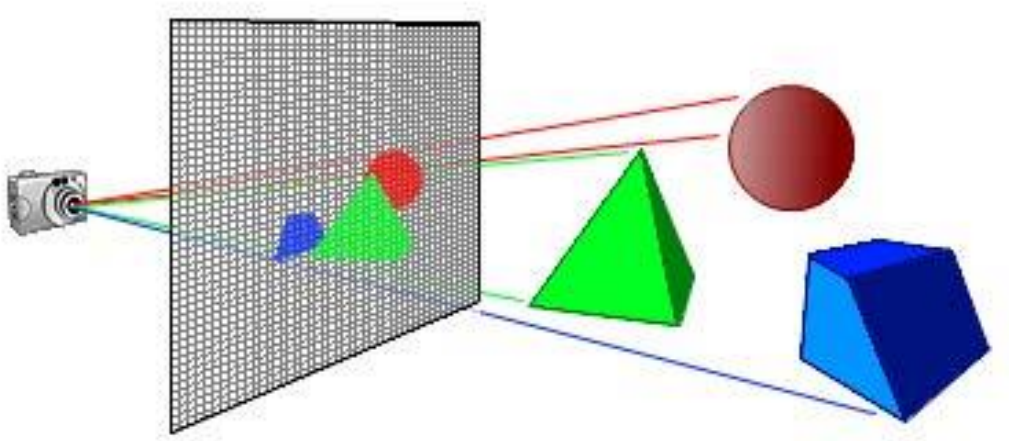
$$c = (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - R^2$$

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



Raycasting

```
for each pixel
   generate a primary ray
   for each object in the scene
      test the intersection between the object and the ray
      if distance is less than the other tests
         remember the intersection point p
         remember the distance to the intersected object d
         remember the normal of the intersected object N
```



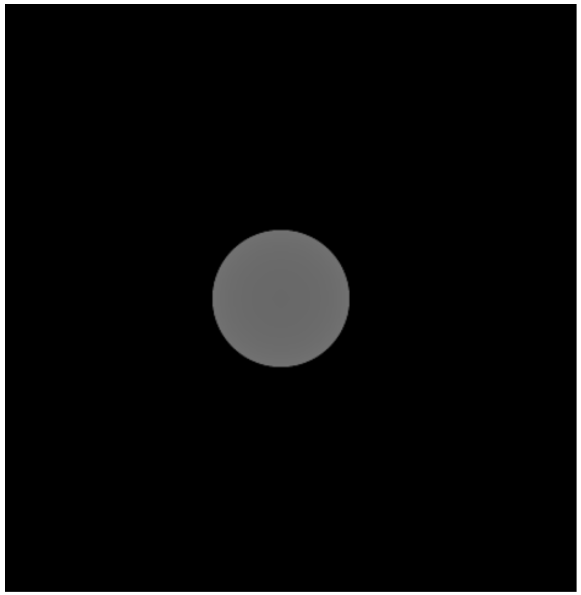
Your work

Define a scene containing

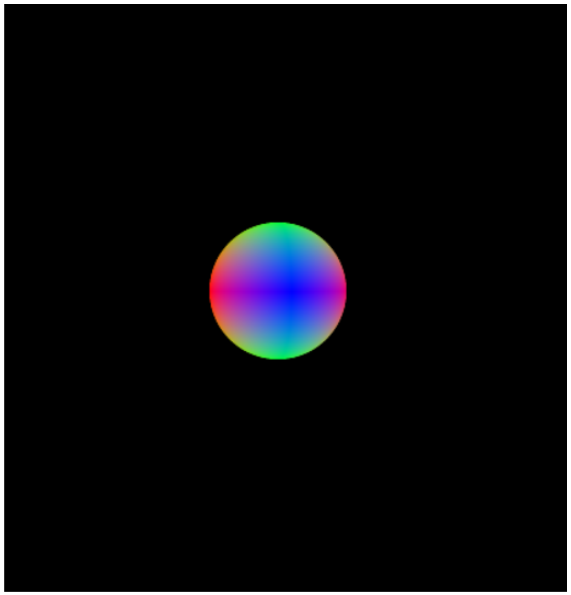
- A camera defining a view
- A sphere

Implement the raycasting algorithm in the fragment shader.

- Send the scene data to the shader via uniform variables
- Generate a primary ray
- Find intersection of the ray with the sphere
- Render depths or normals (we do not consider light sources yet)



Depths



Normals

Fragment Shader

```
uniform float view[16];
uniform float sphere[16];
vec2 UV; // the pixel UV coordinate
out vec4 fColor; // final color

void computePrimaryRay(in vec2 UV, out vec3 rayPos, out vec3 rayDir) { ... }

float raySphere(vec3 rayPos, vec3 rayDir, vec3 spherePos, float sphereRadius, out vec3 intersecPt, out vec3 normal)
{ ... } // test ray-sphere intersection, if intersect: return distance, point and normal

float computeIntersection(vec3 rayPos, vec3 rayDir, out vec3 intersecI, out vec3 normalI)
{ ... } // find intersection with the sphere, if intersect: return distance, point and normal

vec2 computeUV() { ... }
vec3 raycast(vec2 UV) { ... } // compute primary ray, computeIntersection, shade, return color

void main()
{
    UV = computeUV();
    fColor = vec4(raycast(UV),1.0);
}
```

Ray - Plane

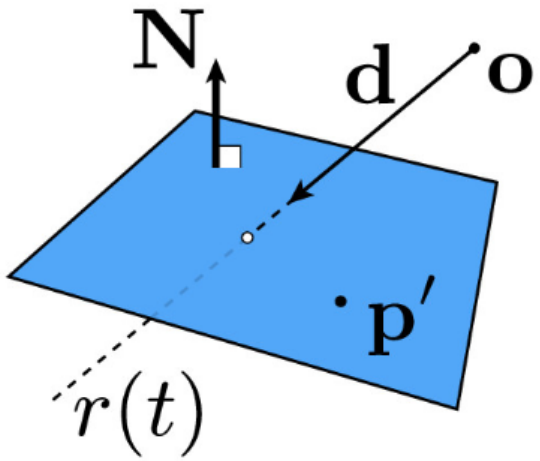
A plane is defined by offset p' and normal N.

Ray equation:

$$\mathbf{r}(t) = \mathbf{o} + t\,\mathbf{d},\ 0 \le t < \infty$$

Plane equation:

$$\mathbf{p} : (\mathbf{p} - \mathbf{p}') \cdot \mathbf{N} = 0$$



Solve for intersection

Set $\mathbf{p} = \mathbf{r}(t)$ and solve for t

$$(\mathbf{p} - \mathbf{p}') \cdot \mathbf{N} = (\mathbf{o} + t\,\mathbf{d} - \mathbf{p}') \cdot \mathbf{N} = 0$$

$$t = \frac{(\mathbf{p}' - \mathbf{o}) \cdot \mathbf{N}}{\mathbf{d} \cdot \mathbf{N}}$$

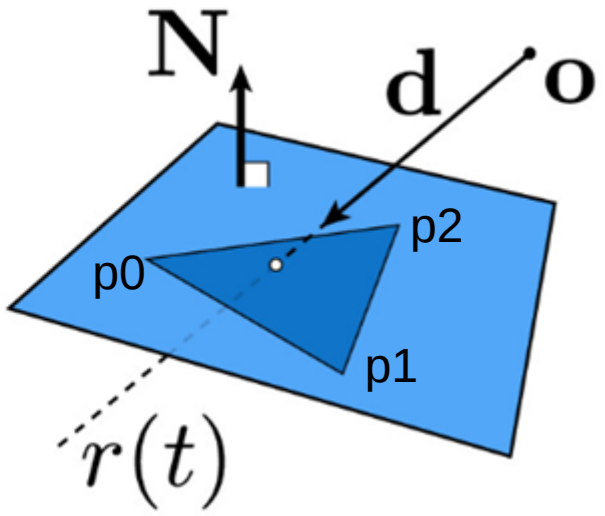
Check: $0 \le t < \infty$

Ray - Triangle

A triangle is defined by three vertices p0, p1, p2.

Triangle is in a plane

- **Ray-plane intersection**
- **Test if hit point is inside triangle**
  (inside-outside test or barycentric coordinates test)



Ray - Triangle

Möller and Trumbore proposed in 1997 a fast algorithm for calculating the intersection of a ray and a triangle in three dimensions without needing the precomputation of the plane equation

$$\vec{\mathbf{O}} + t\vec{\mathbf{D}} = (1 - b_1 - b_2)\vec{\mathbf{P}}_0 + b_1\vec{\mathbf{P}}_1 + b_2\vec{\mathbf{P}}_2$$

Where:

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{\vec{\mathbf{S}}_1 \bullet \vec{\mathbf{E}}_1} \begin{bmatrix} \vec{\mathbf{S}}_2 \bullet \vec{\mathbf{E}}_2 \\ \vec{\mathbf{S}}_1 \bullet \vec{\mathbf{S}} \\ \vec{\mathbf{S}}_2 \bullet \vec{\mathbf{D}} \end{bmatrix}$$

$$\vec{\mathbf{E}}_1 = \vec{\mathbf{P}}_1 - \vec{\mathbf{P}}_0$$

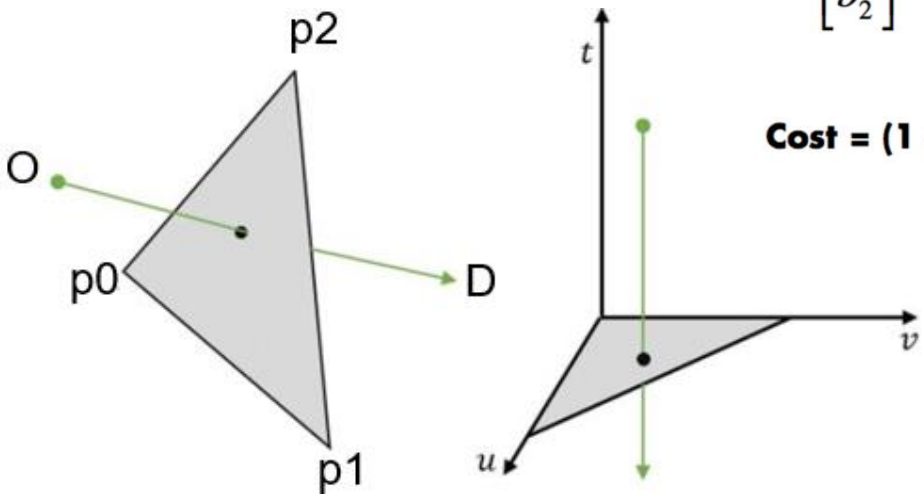
$$\vec{\mathbf{E}}_2 = \vec{\mathbf{P}}_2 - \vec{\mathbf{P}}_0$$

$$\vec{\mathbf{S}} = \vec{\mathbf{O}} - \vec{\mathbf{P}}_0$$

$$\vec{\mathbf{S}}_1 = \vec{\mathbf{D}} \times \vec{\mathbf{E}}_2$$

$$\vec{\mathbf{S}}_2 = \vec{\mathbf{S}} \times \vec{\mathbf{E}}_1$$

Cost = (1 div, 27 mul, 17 add)



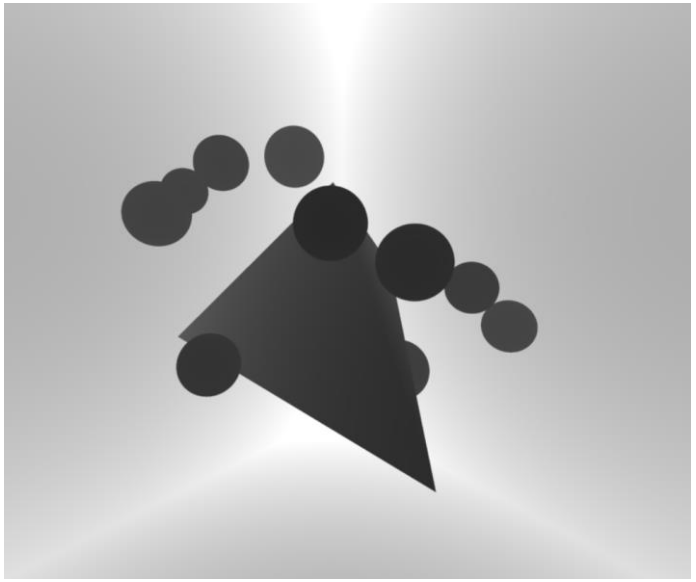
Your work

Define a scene containing

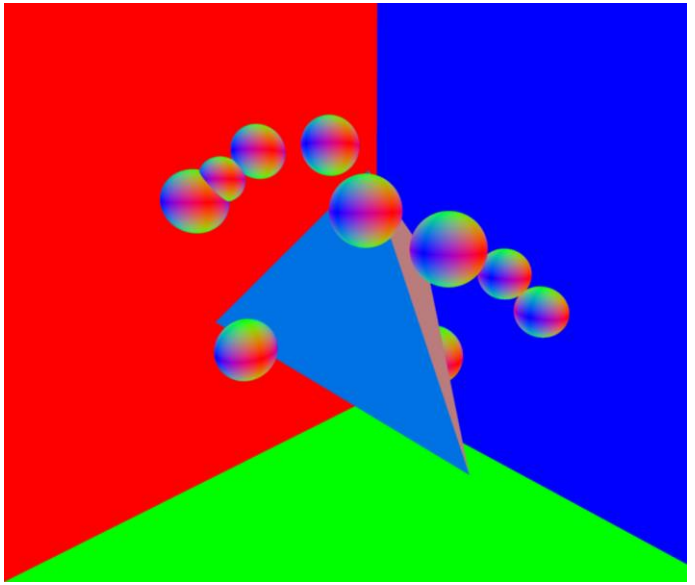
- A movable camera defining a view
- At least one sphere
- At least one plane
- At least one handcrafted triangle mesh
- A complex triangle mesh loaded from an obj file into a uniform buffer object [BONUS]

Implement the raycasting algorithm in the fragment shader.

- Send the scene data to the shader via uniform variables
- Generate a primary ray
- Find closest intersection of the ray with the scene
- Render depths and normals (we do not consider light sources yet)



Depths



Normals

Fragment Shader

```
uniform int nbSpheres;
uniform int nbPlanes;
uniform int nbMeshTriangles;
uniform float view[16];
uniform float sphere[256];
uniform float plane[256];
uniform float mesh[256];
vec2 UV; // the pixel UV coordinate
out vec4 fColor; // final color

void computePrimaryRay(in vec2 UV, out vec3 rayPos, out vec3 rayDir) { ... }

float rayTriangle(vec3 rayPos, vec3 rayDir, vec3 p0, vec3 p1, vec3 p2, out vec3 intersecPt, out vec3 normal)
{ ... } // test ray-triangle intersection, if intersect: return distance, point and normal

float raySphere(vec3 rayPos, vec3 rayDir, vec3 spherePos, float sphereRadius, out vec3 intersecPt, out vec3 normal)
{ ... } // test ray-sphere intersection, if intersect: return distance, point and normal

float rayPlane(vec3 rayPos, vec3 rayDir, vec3 planePos, vec3 planeNormal, out vec3 intersecPt, out vec3 normal)
{ ... } // test ray–plane intersection , if intersect : return distance, point and normal

float computeNearestIntersection(vec3 rayPos, vec3 rayDir, out vec3 intersecI, out vec3 normalI)
{ ... } // find nearest intersection in the scene, , if intersect: return distance, point and normal

vec2 computeUV() { ... }
vec3 raycast(vec2 UV) { ... } // compute primary ray, computeNearestIntersection, shade, return color

void main()
{
    UV = computeUV();
    fColor = vec4(raycast(UV),1.0);
}
```