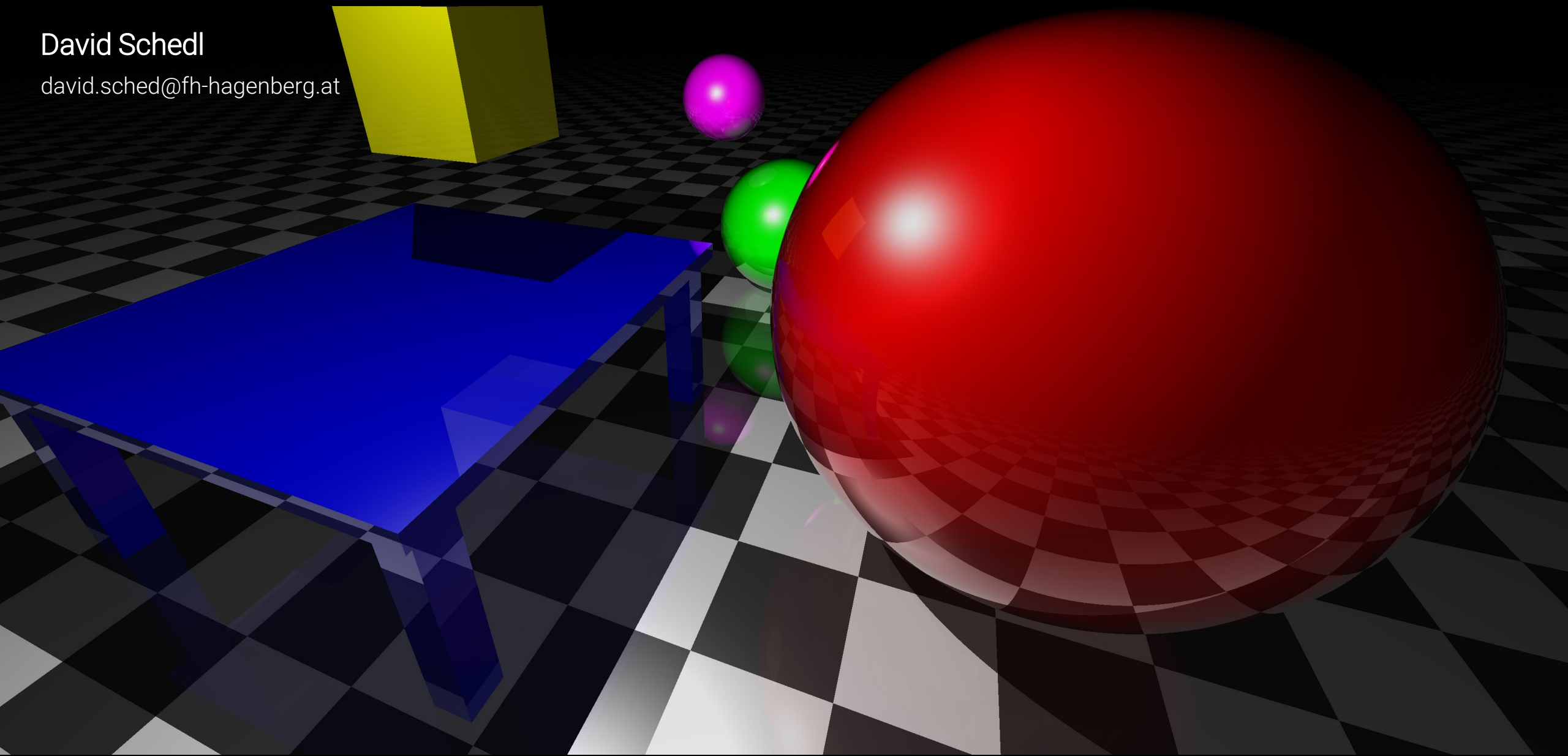
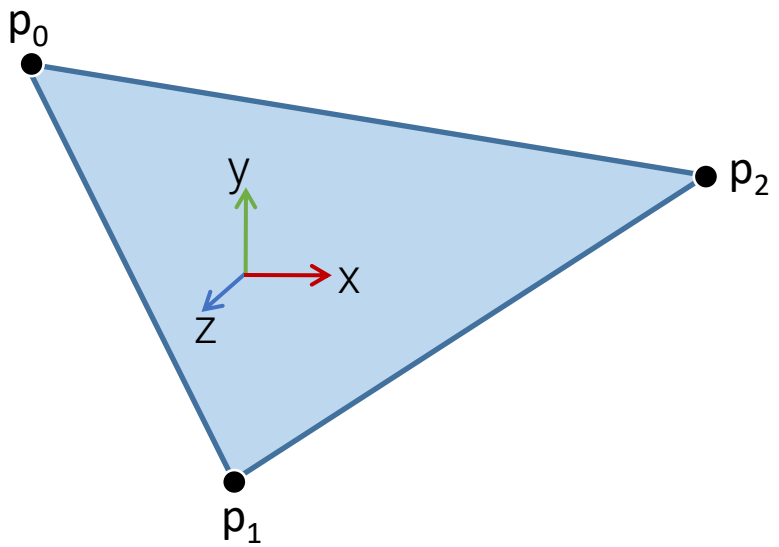
Raytracing

David Schedl

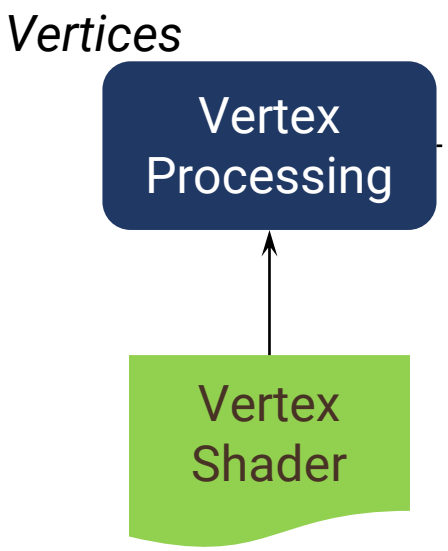
david.sched@fh-hagenberg.at

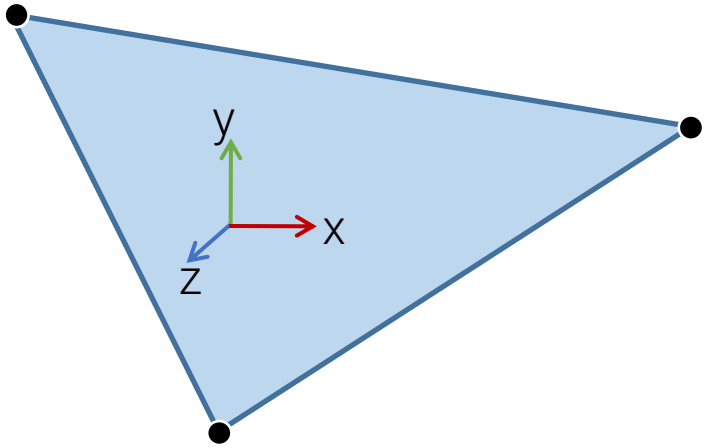


Recap Rasterization

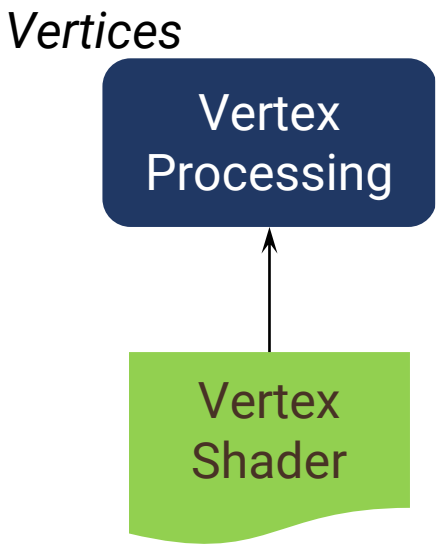


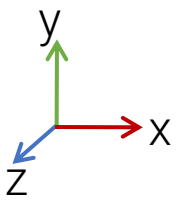
Recap Rasterization

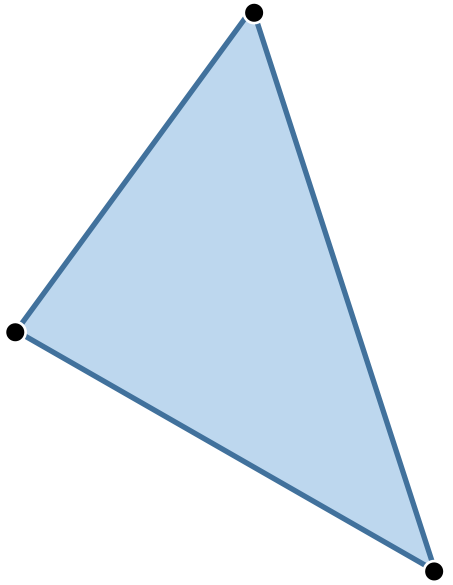




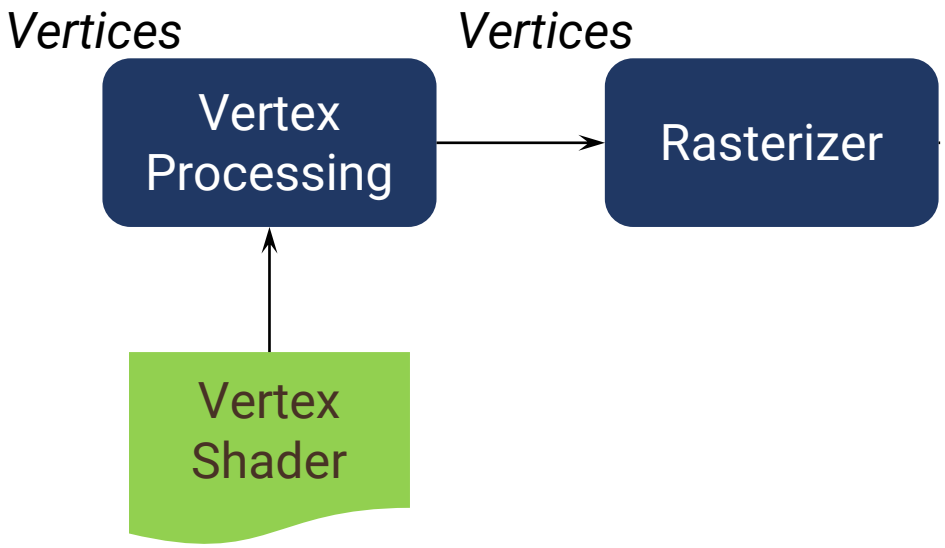
Recap Rasterization

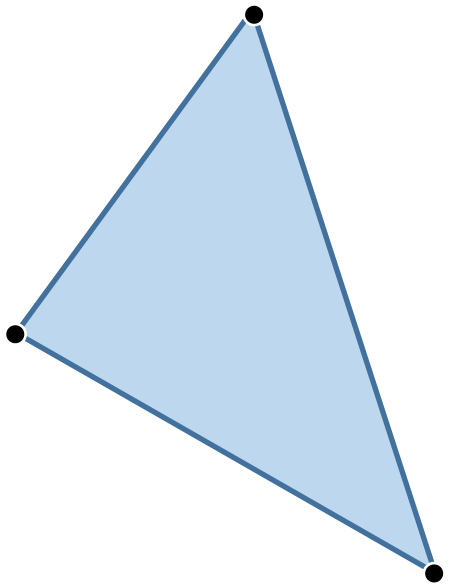




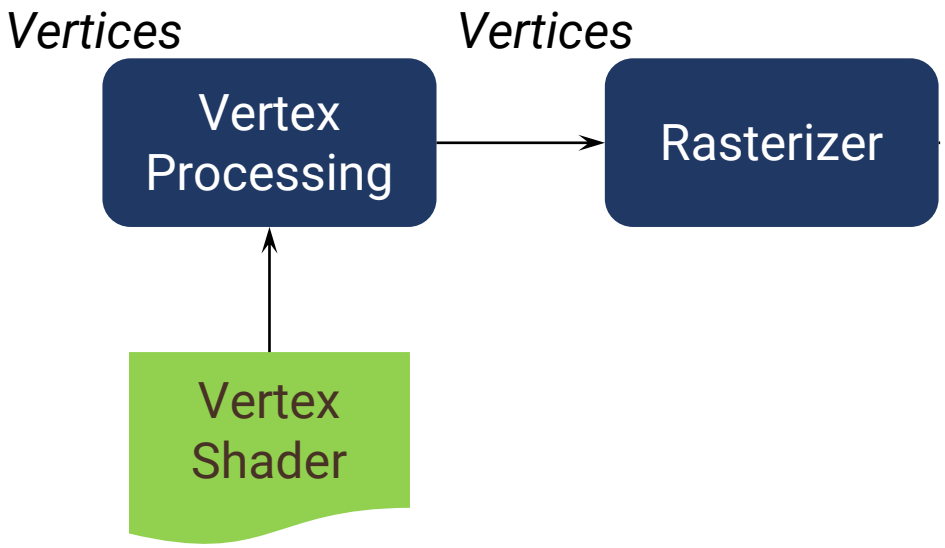


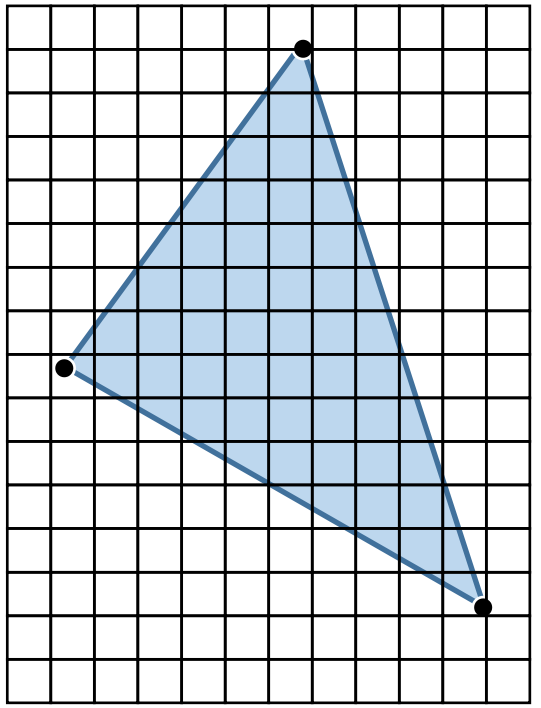
Recap Rasterization



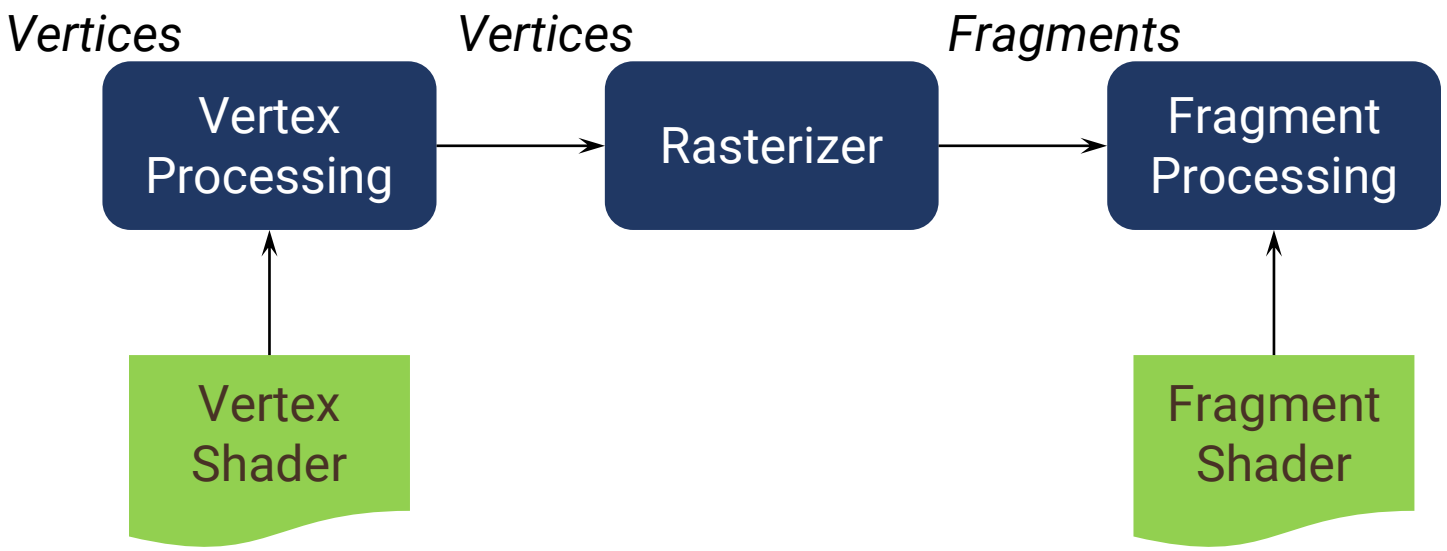


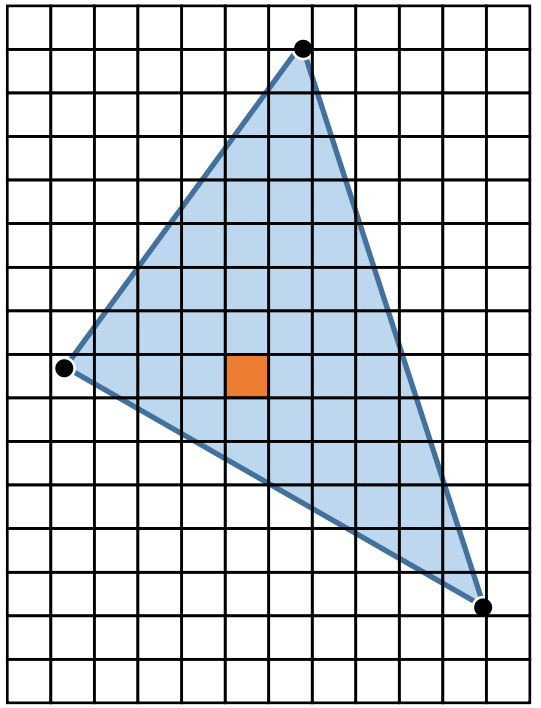
Recap Rasterization



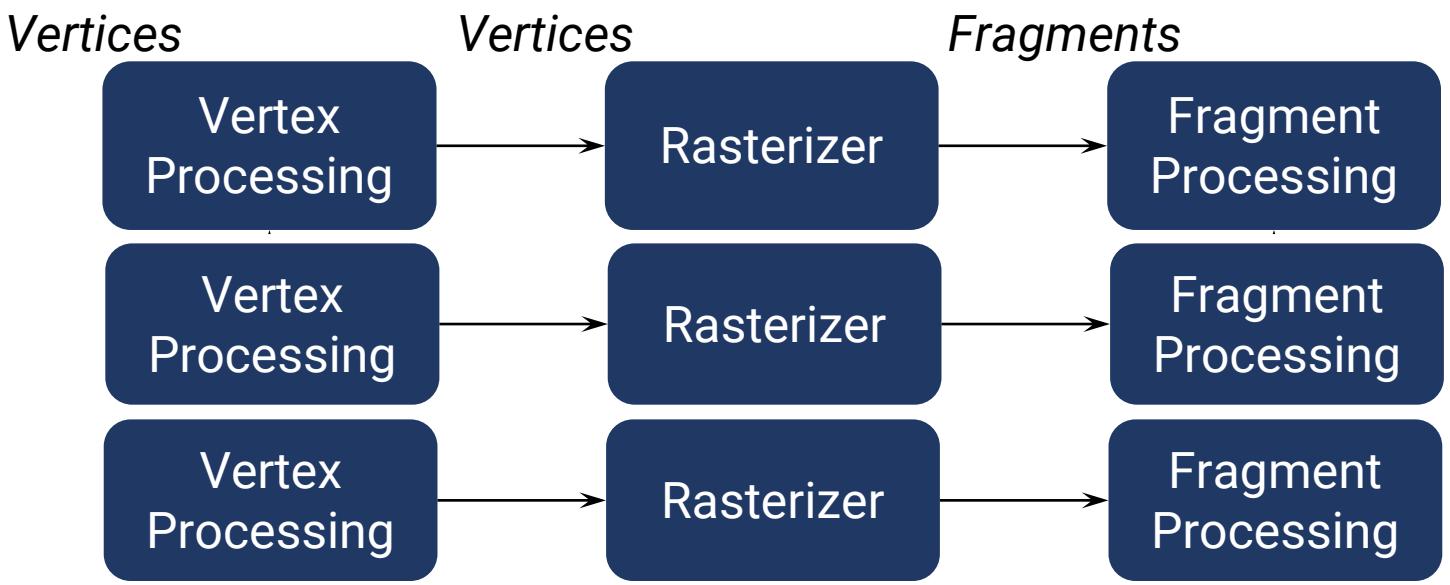


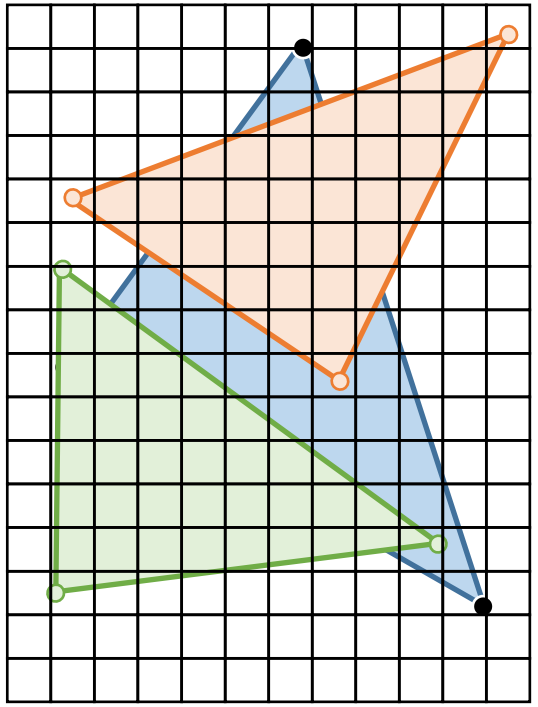
Recap Rasterization





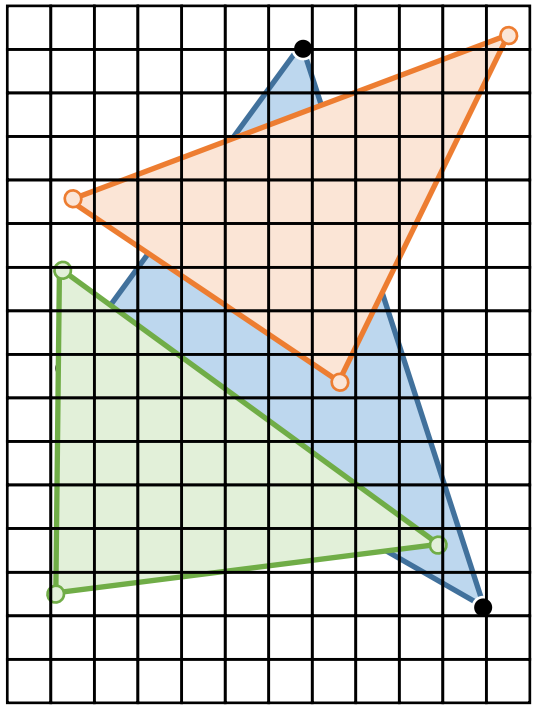
Recap Rasterization





Recap Rasterization

- Rasterization pipeline runs
  - parallel (vertices and fragments) and
  - independent for every mesh.
- No information about the rest of the scene.
- Raytracing:
  - completely different idea!

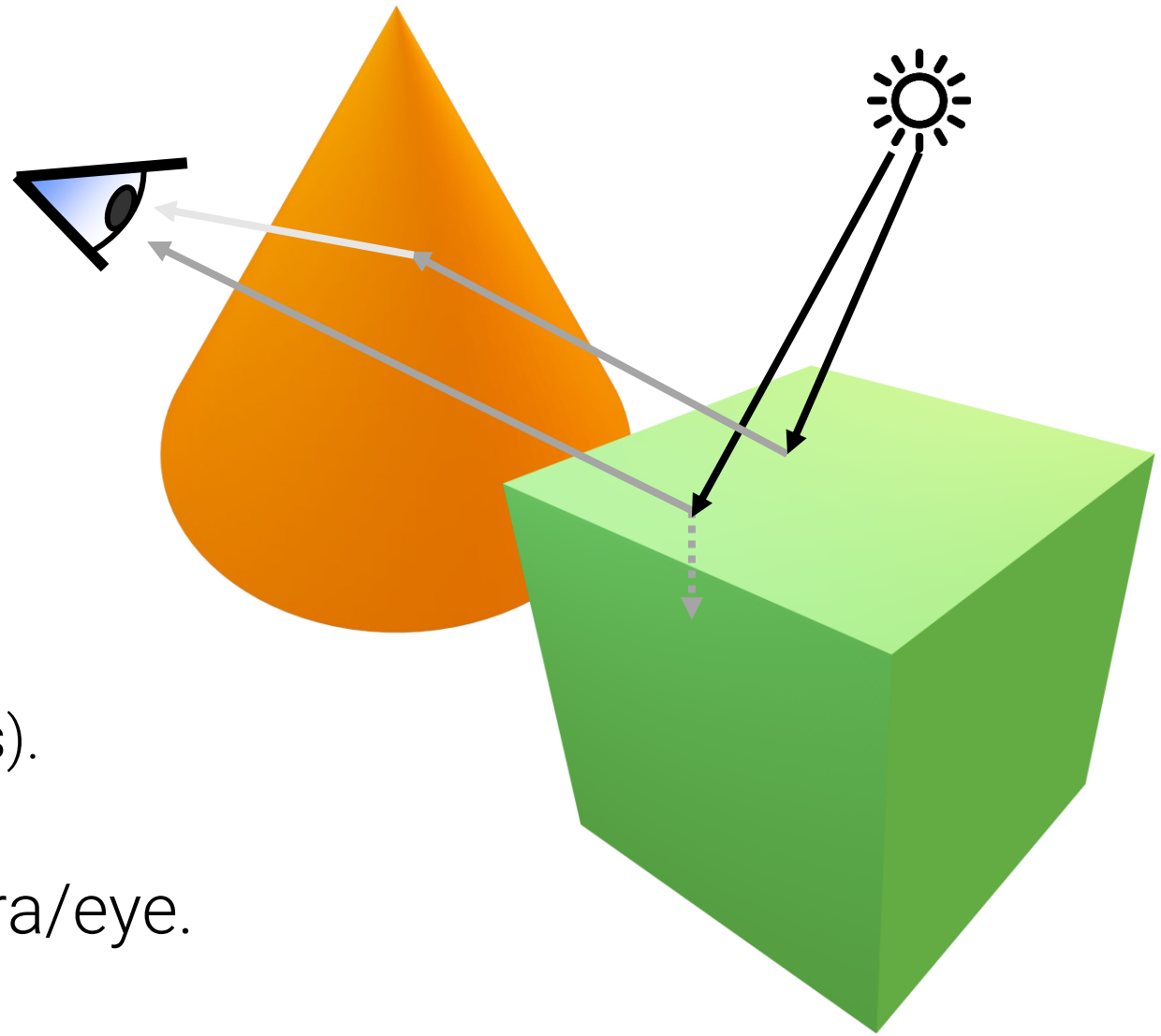


Raytracing

- Light rays interact with the scene.

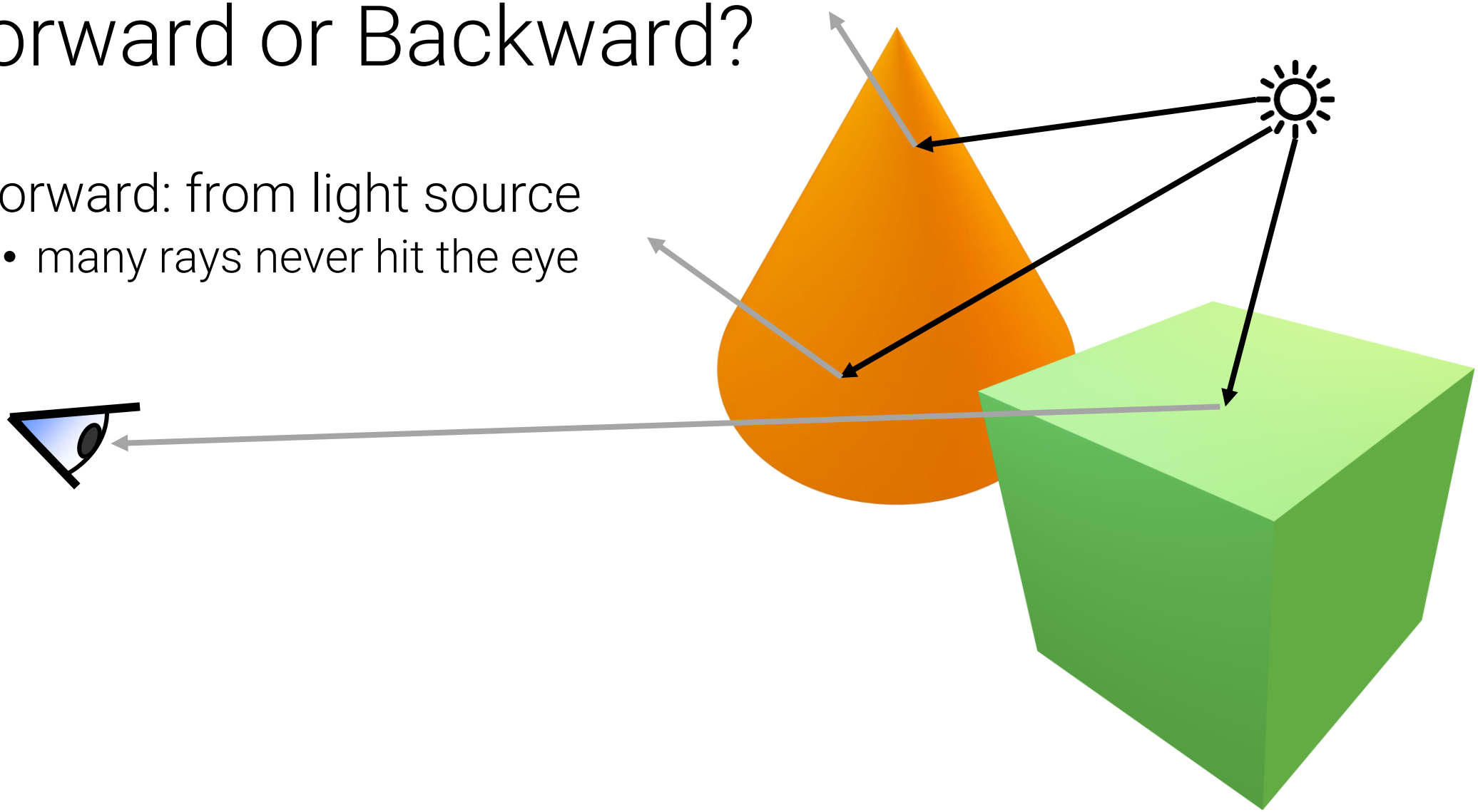
- Hitting an Object:
  - reflect and
  - refract
  - and shoot new ray (bounces).

- Trace until it hits the camera/eye.



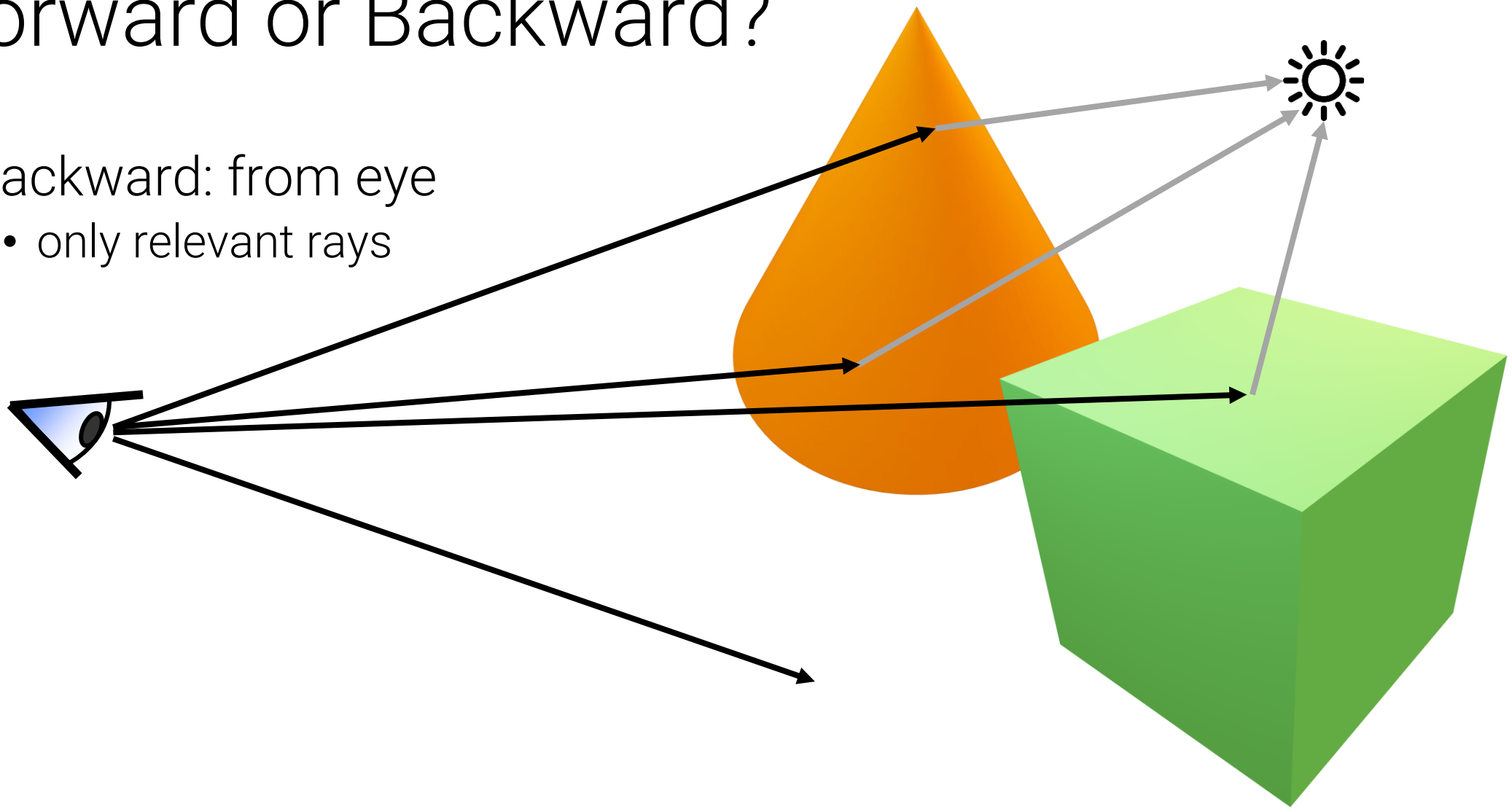
Forward or Backward?

- Forward: from light source
  - many rays never hit the eye



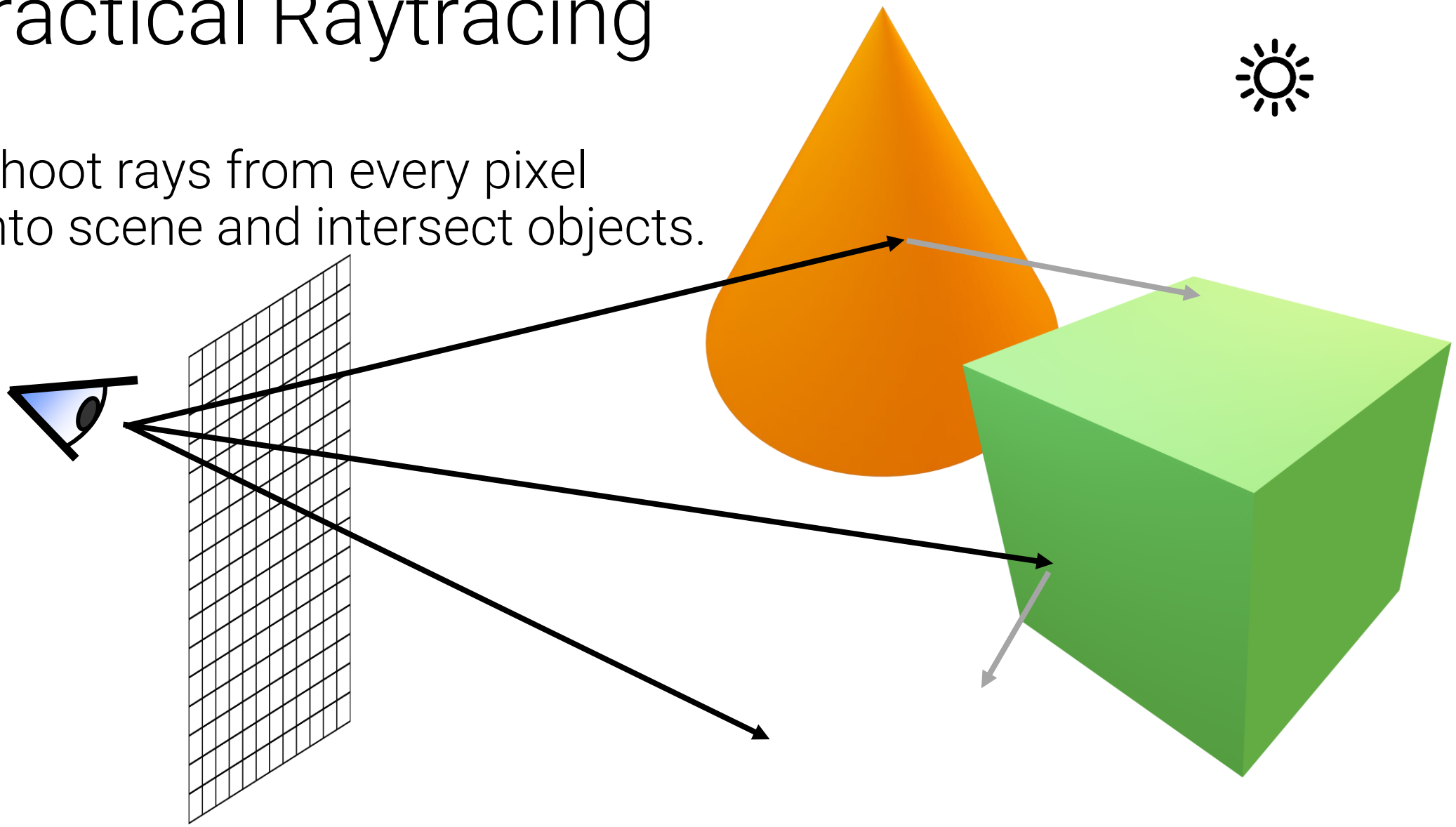
Forward or Backward?

- Backward: from eye
  - only relevant rays



Practical Raytracing

- Shoot rays from every pixel into scene and intersect objects.

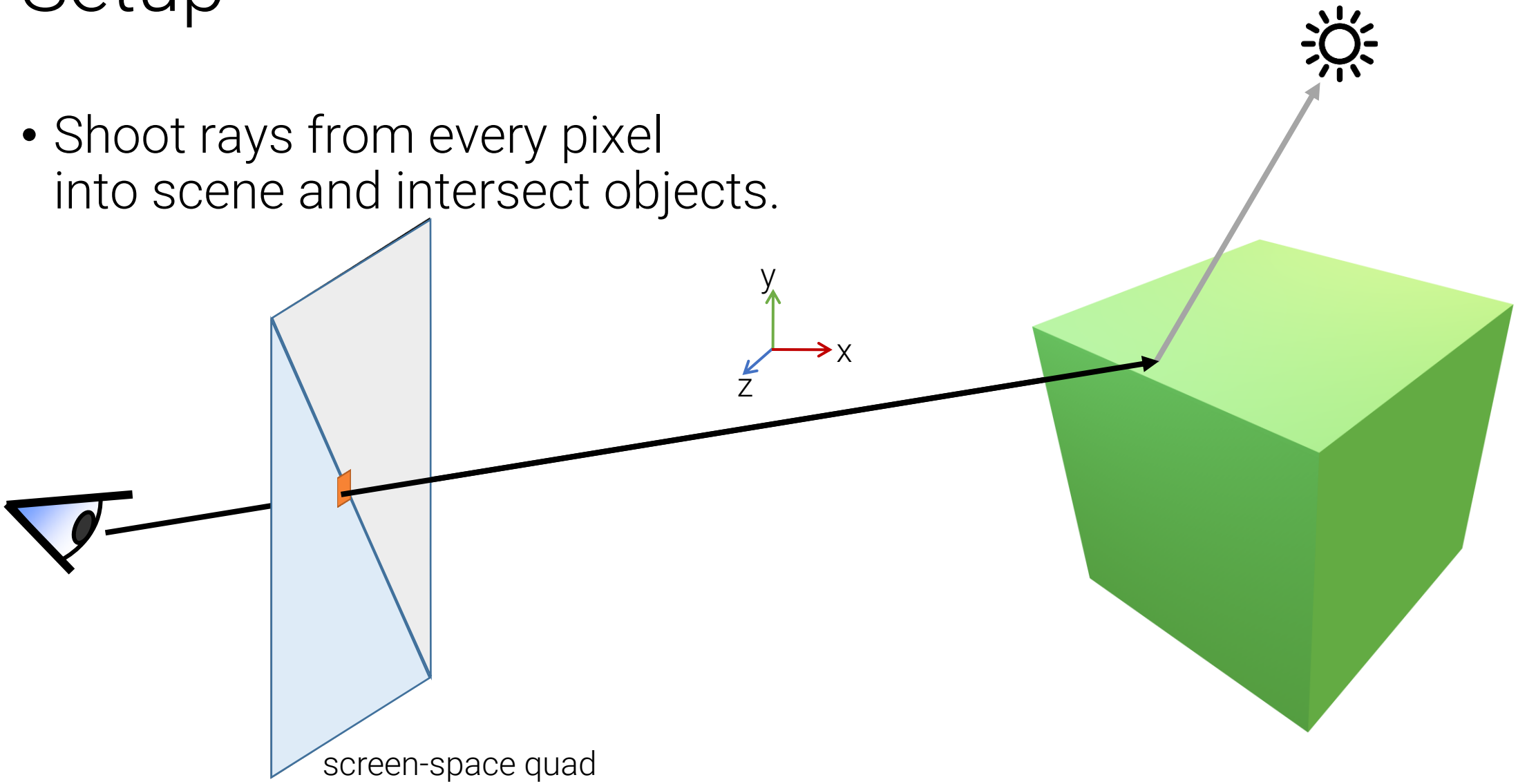


TASK: Raytracing in GLSL

- Download the source code.
- Let’s get raytracing working.
- Hints:
  - We only edit the FS!

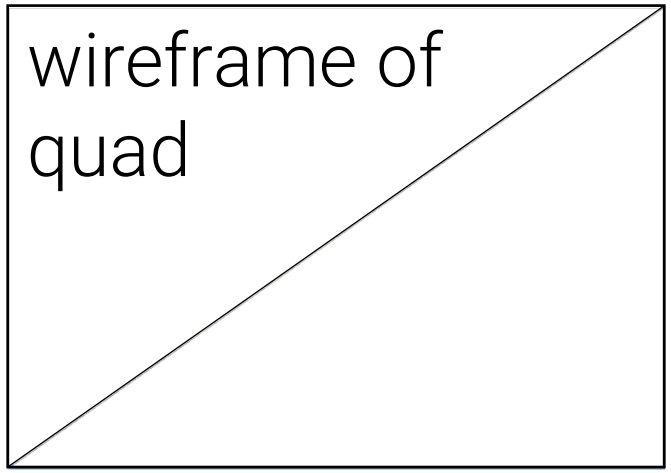
Setup

- Shoot rays from every pixel into scene and intersect objects.



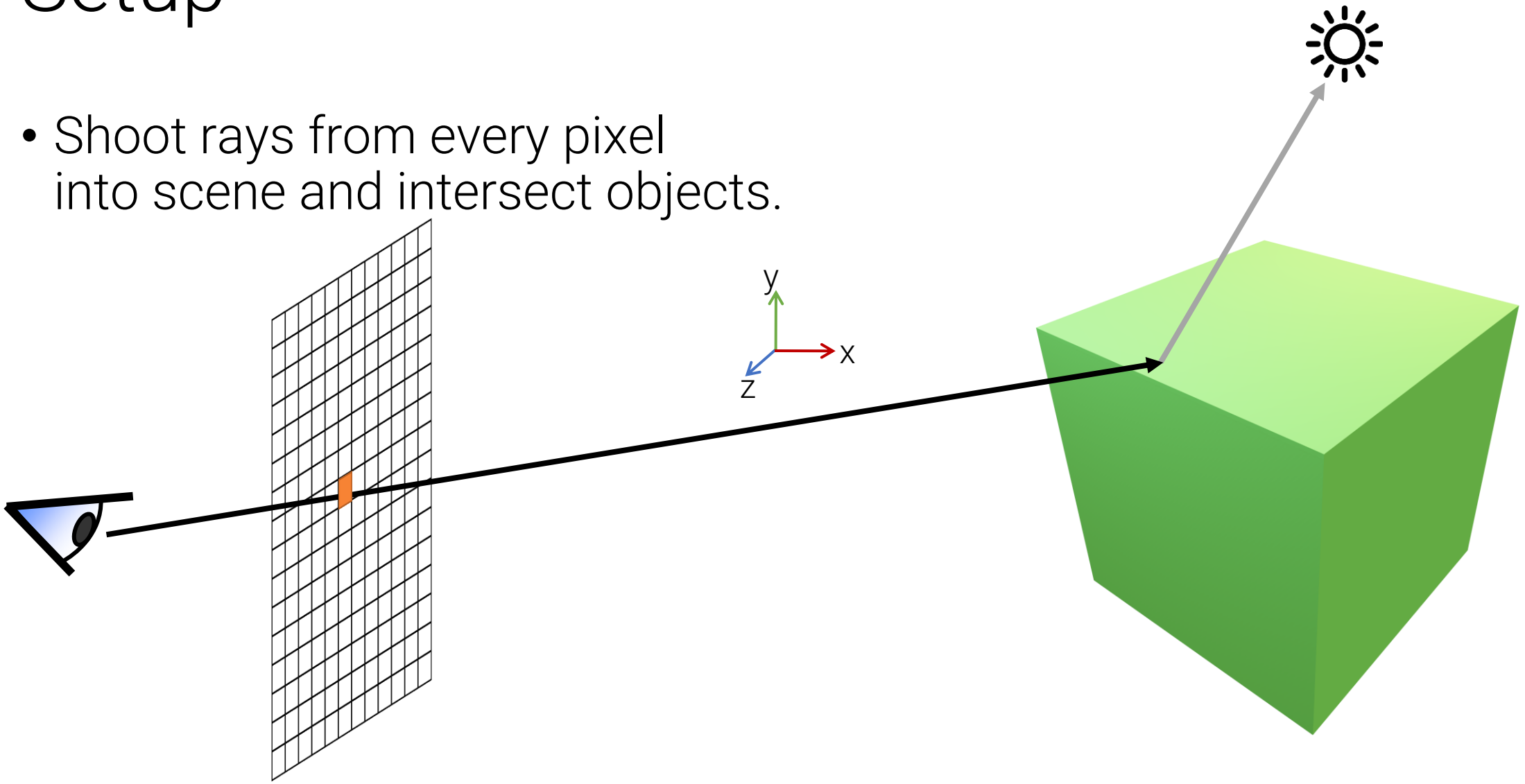
Code

```
...
float quadVertices[] = {
            // positions         // texture Coords
            -1.0f,  1.0f, 0.0f, 0.0f, 1.0f,
            -1.0f, -1.0f, 0.0f, 0.0f, 0.0f,
             1.0f,  1.0f, 0.0f, 1.0f, 1.0f,
             1.0f, -1.0f, 0.0f, 1.0f, 0.0f,
        };
```



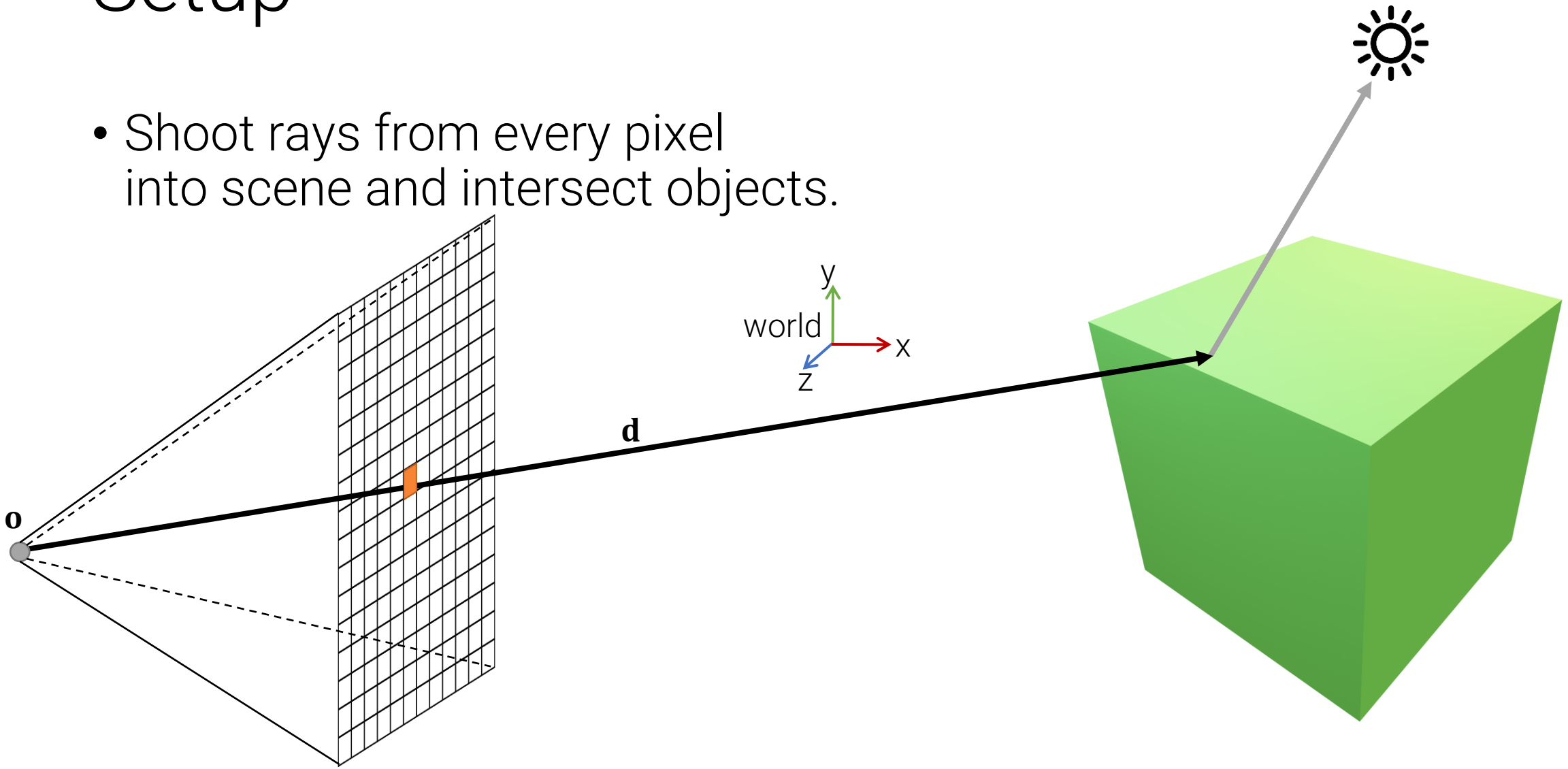
Setup

- Shoot rays from every pixel into scene and intersect objects.



Setup

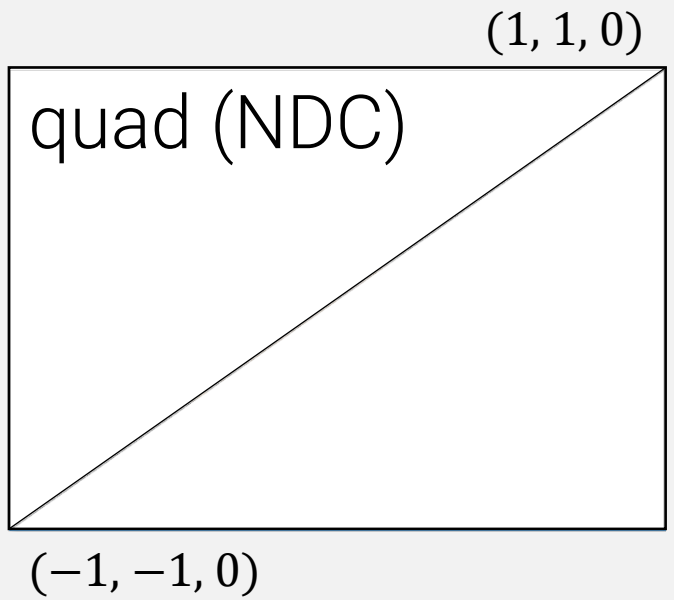
- Shoot rays from every pixel into scene and intersect objects.



The Vertex Shader

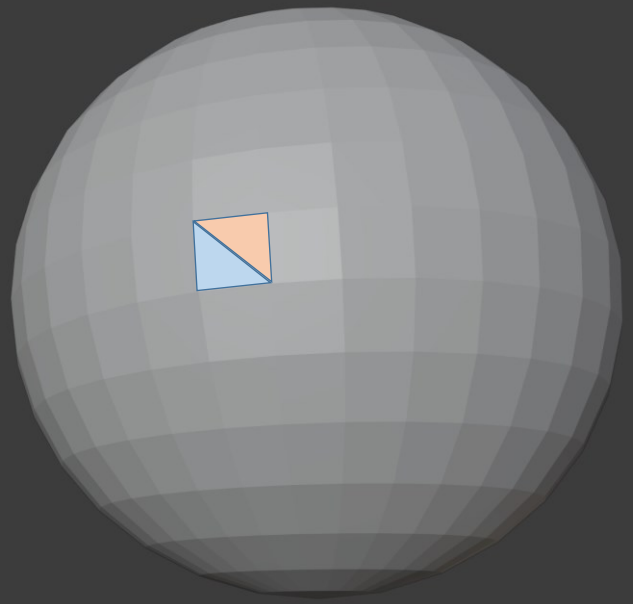
raytracing.vs.glsl

```
…
//output of this shader
out vec3 upOffset;
out vec3 rightOffset;
out vec3 rayOrigin;
out vec3 rayDir;

void main() {

    // positions in world space:
    rayOrigin = camPos; // ray origin (o)
    mat4 invView = inverse(view);
    vec4 worldPos = inverse(projection*view) * vec4( a_position.xyz, 1.0 );
    rayDir = worldPos.xyz/worldPos.w - rayOrigin; // ray direction (d)
    gl_Position = vec4(a_position, 1.0);

}
```

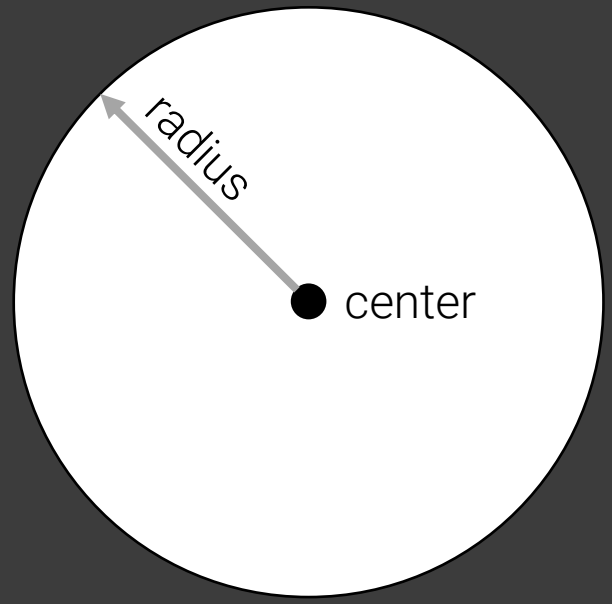


Scene Objects

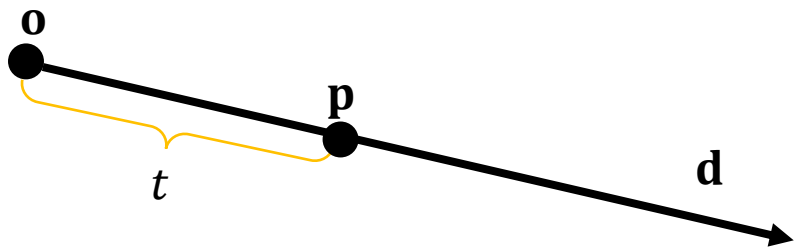
Triangle Mesh (Rasterization)



Implicit Object (Raytracing)



Ray-object intersections



- The parametric ray equation defines a point on the ray by

  $\mathbf{p} = \mathbf{o} + t\mathbf{d}$ with ray origin $\mathbf{o} = \begin{pmatrix} o_x \\ o_y \\ o_z \end{pmatrix}$, ray direction $\mathbf{d} = \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}$

- Implicit surfaces equations define points on the surface by

  $f(\mathbf{p}) = 0, \quad \mathbf{p} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$

- For a possible (ray-object) intersection point it must hold:

  $f(\mathbf{o} + t\mathbf{d}) = 0$

Ray-sphere intersection

- Implicit sphere (center $\mathbf{c} = (c_x, c_y, c_z)^T$, radius r)

$$0 = (x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 - r^2$$
$$= (\mathbf{x} - \mathbf{c}) \cdot (\mathbf{x} - \mathbf{c}) - r^2$$

- Substitute parametric ray and rearrange:

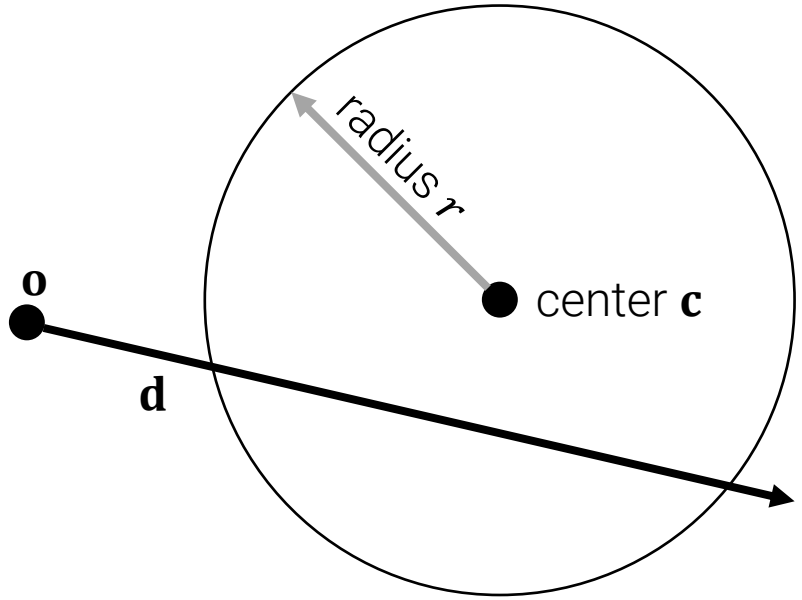
$$(\mathbf{o} + t\vec{\mathbf{d}} - \mathbf{c}) \cdot (\mathbf{o} + t\vec{\mathbf{d}} - \mathbf{c}) - r^2 = 0$$

$$\underbrace{\vec{\mathbf{d}} \cdot \vec{\mathbf{d}}}_{a} t^2 + \underbrace{2\vec{\mathbf{d}} \cdot (\mathbf{o} - \mathbf{c})}_{b} t + \underbrace{(\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2}_{c} = 0$$

⇒ Quadratic equation in t

⇒ Discriminant determines the number of solutions

⇒ No intersection if $b^2 - 4ac < 0$



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

GLSL Code: Ray-sphere inter.

raytracing.fs.glsl

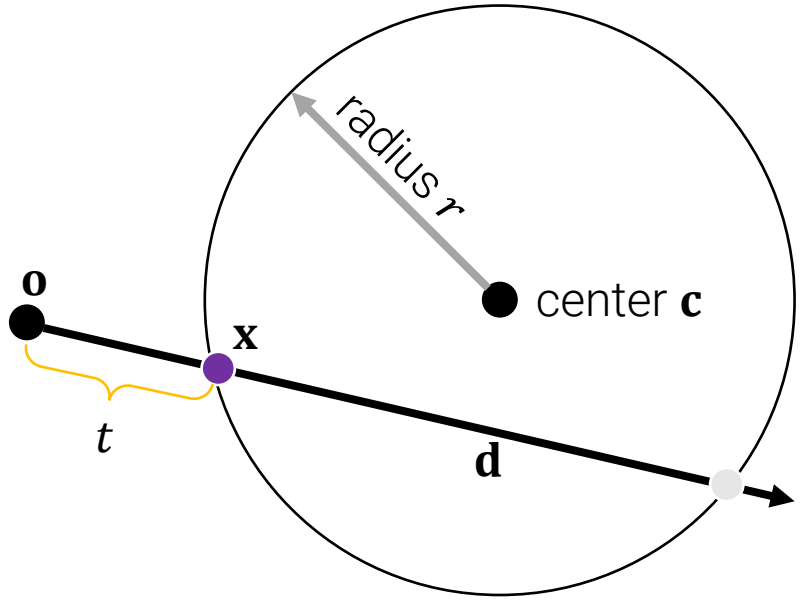
```
float intersectSphere(vec3 origin, vec3 ray, vec3 sphereCenter, float sphereRadius)
{
   vec3 toSphere = origin - sphereCenter;
   float a = dot(ray, ray);
   float b = 2.0 * dot(toSphere, ray);
   float c = dot(toSphere, toSphere) - sphereRadius*sphereRadius;
   float discriminant = b*b - 4.0*a*c;
   if(discriminant > 0.0) {
     float t = (-b - sqrt(discriminant)) / (2.0 * a);
     if(t > 0.0) { return t; }
   }
   return INFINITY ;
}
```

Ray-sphere intersections (2)

- Hit point:

$$\mathbf{x} = \mathbf{o} + t\vec{\mathbf{d}}$$

- Normal of hit point?



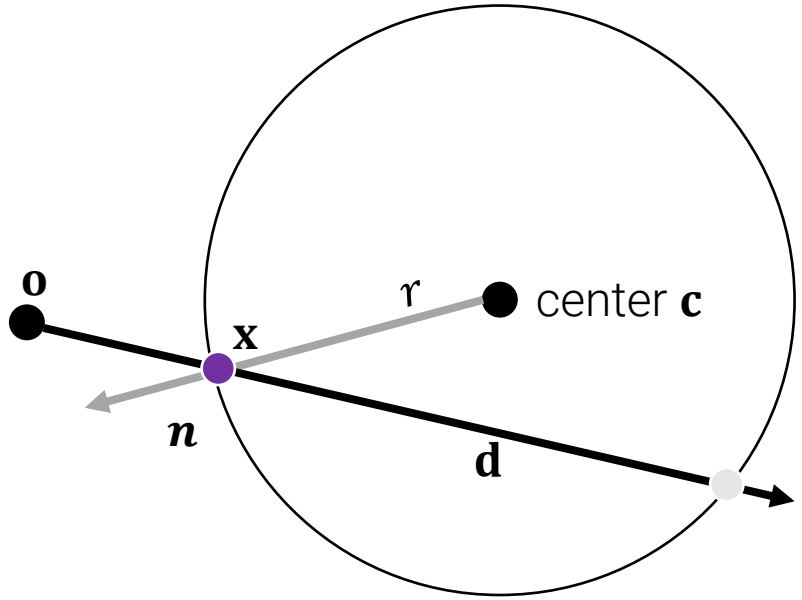
Ray-sphere intersections (2)

- Hit point:

$$\mathbf{x} = \mathbf{o} + t\vec{\mathbf{d}}$$

- Normal of hit point:

$$\boldsymbol{n} = (\mathbf{x} - \mathbf{c})/r$$



GLSL Code:

```
vec3 normalForSphere(vec3 hit, vec3 sphereCenter, float sphereRadius) {
   return (hit - sphereCenter) / sphereRadius;
}
```

raytracing.fs.glsl

GLSL Code: Sphere

raytracing.fs.glsl

```
void addSphere( vec3 ro, vec3 rd, vec3 center, float radius, vec3 color,
   inout float hitDist, inout vec3 hitColor, inout vec3 hitNormal )
{
  float dist = intersectSphere(ro, rd, center, radius);

  if (dist < hitDist) {
    vec3 hit  = ro + dist * rd;
    hitNormal = normalForSphere(hit, center, radius);
    hitColor  = color;
    hitDist   = dist;
  }
}
```

GLSL Code: Sphere

raytracing.fs.glsl

```
void addSphere( vec3 ro, vec3 rd, vec3 center, float radius, vec3 color,
   inout float hitDist, inout vec3 hitColor, inout vec3 hitNormal )
{
  float dist = intersectSphere(ro, rd, center, radius);

  if (dist < hitDist) {
    vec3 hit  = ro + dist * rd;
    hitNormal = normalForSphere(hit, center, radius);
    hitColor  = color;
    hitDist   = dist;
  }
}
```

- output: distance (to compute hit point), color, normal (for shading and reflections)

GLSL Code: Sphere

raytracing.fs.glsl

```
void addSphere( vec3 ro, vec3 rd, vec3 center, float radius, vec3 color,
   inout float hitDist, inout vec3 hitColor, inout vec3 hitNormal )
{
  float dist = intersectSphere(ro, rd, center, radius);

  if (dist < hitDist) {
    vec3 hit   = ro + dist * rd;
    hitNormal = normalForSphere(hit, center, radius);
    hitColor  = color;
    hitDist   = dist;
  }
}
```

- check if closest intersection!

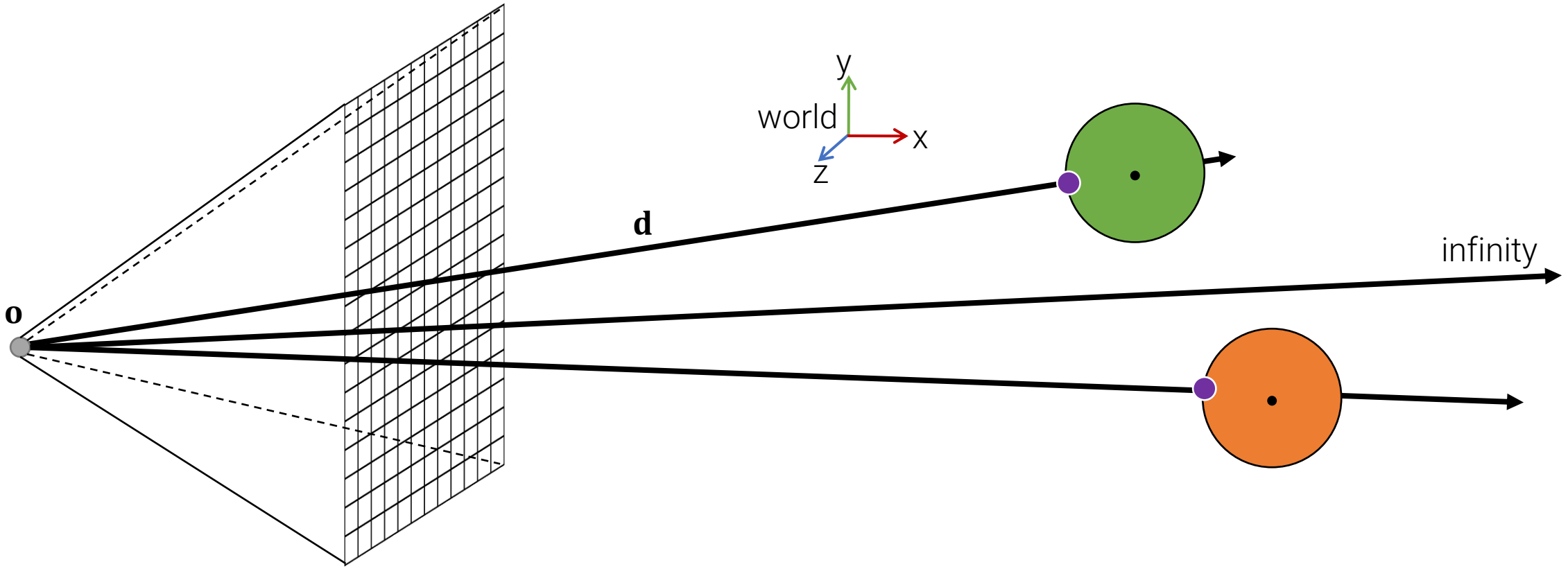
Scene Setup

raytracing.fs.glsl

```
float rayTraceScene(vec3 ro /*rayStart*/, vec3 rd /*rayDirection*/, out vec3
        hitNormal, out vec3 hitColor)
{
  float hitDist = INFINITY;
  hitNormal = vec3(0.0,0.0,0.0);
  // 3 spheres
  addSphere( ro, rd, vec3(1.0, 2.0, 5.0), 2.0, vec3(1.0, 0.0, 0.0), hitDist,
       hitColor, hitNormal  );
  addSphere( ro, rd, vec3(5.0, 1.0, 2.0), 1.0, vec3(0.0, 1.0, 0.0), hitDist,
       hitColor, hitNormal  );
  addSphere( ro, rd, vec3(5.0, 2.8, 1.0), 0.6, vec3(1.0, 0.0, 1.0), hitDist,
       hitColor, hitNormal );
  …
  return hitDist;
}
```

One Ray per Fragment

- Shoot ray (from every fragment) into scene and intersect objects.

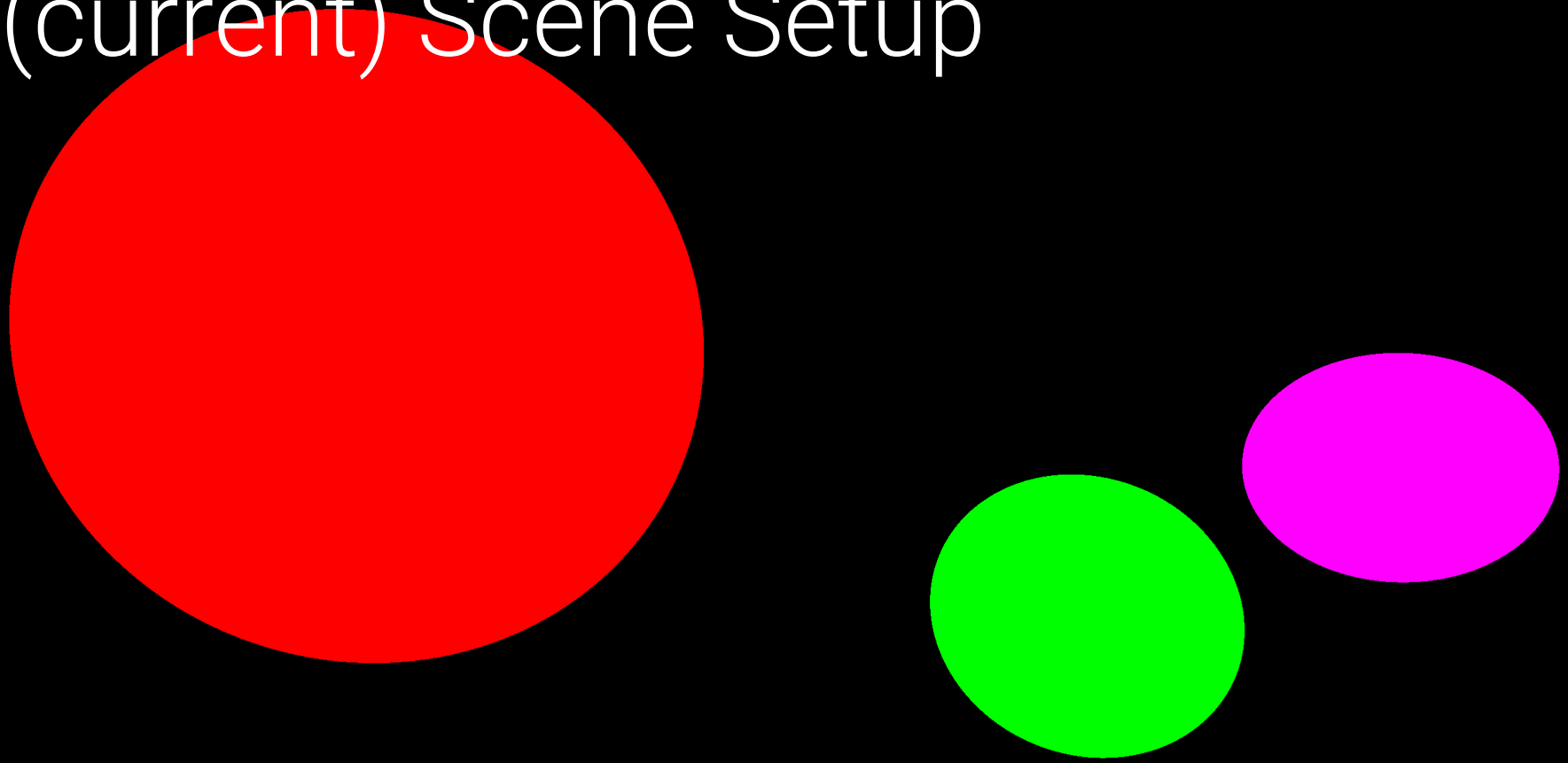


One Ray per Fragment

raytracing.fs.glsl

```
void main() {
  gl_FragColor.rgba = vec4(0.0, 0.0, 0.0, 1.0);
  vec3 rayStart = rayOrigin;
  vec3 rayDirection = normalize(rayDir);

  vec3 hitColor = vec3( 0.0 );
  vec3 hitNormal;
  float dist = rayTraceScene(rayStart, rayDirection, hitNormal, hitColor);

  gl_FragColor.rgb = hitColor;
}
```

Our (current) Scene Setup



```
…
  addSphere( ro, rd, vec3(1.0, 2.0, 5.0), 2.0, vec3(1.0, 0.0, 0.0), hitDist, hitColor, hitNormal );
  addSphere( ro, rd, vec3(5.0, 1.0, 2.0), 1.0, vec3(0.0, 1.0, 0.0), hitDist, hitColor, hitNormal );
  addSphere( ro, rd, vec3(5.0, 2.8, 1.0), 0.6, vec3(1.0, 0.0, 1.0), hitDist, hitColor, hitNormal );
…
```

Ray-plane intersections

- Plane equation $\mathbf{n} \cdot \mathbf{x} + d = 0$

  plane normal $\mathbf{n} = \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix}$ with $\|\mathbf{n}\| = 1$, distance to origin d

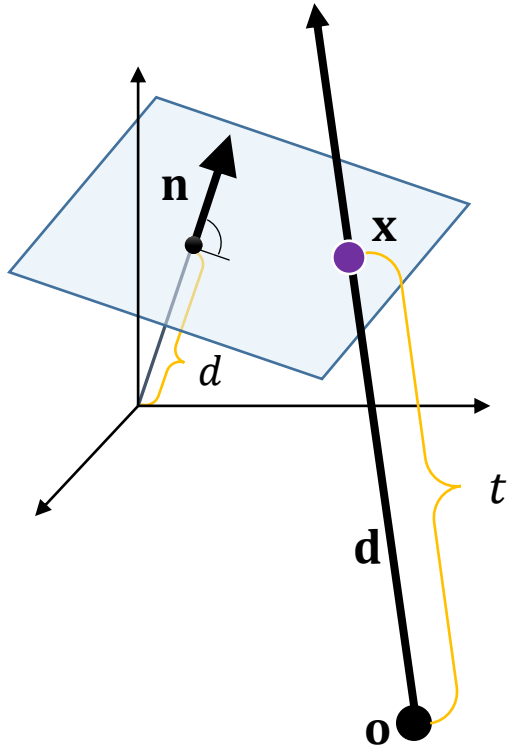
- Substitue parametric ray equation

$$0 = \mathbf{n} \cdot (\mathbf{o} + t\mathbf{d}) + d$$

$$= (\mathbf{n} \cdot \mathbf{o}) + t(\mathbf{n} \cdot \mathbf{d}) + d \quad \Rightarrow \quad t = \frac{-d - \mathbf{n} \cdot \mathbf{o}}{\mathbf{n} \cdot \mathbf{d}}$$

Check the denominator $|\mathbf{n} \cdot \mathbf{d}| < \epsilon$ for ray ∥ plane

Hit if $t > 0$

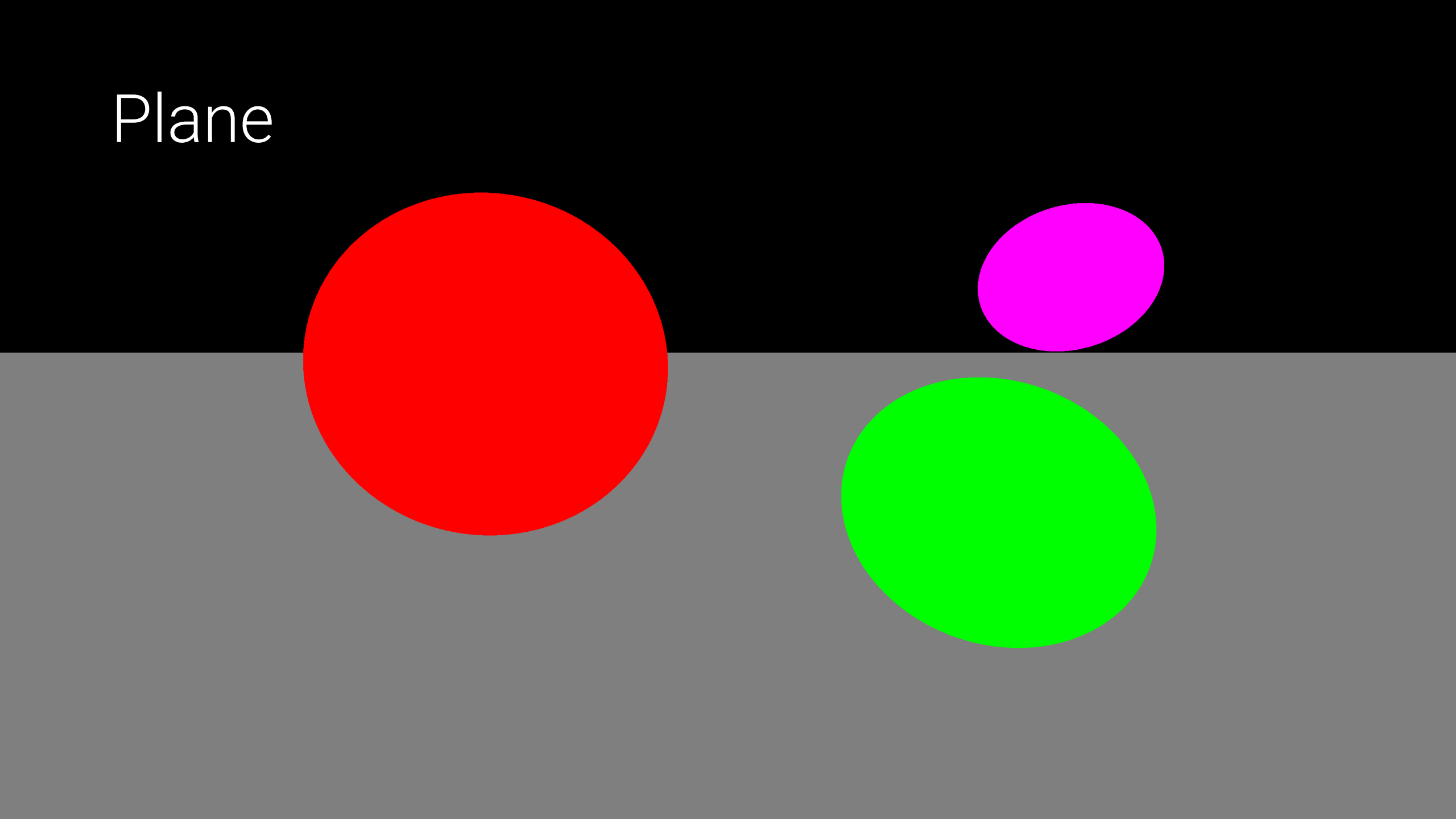


GLSL Code: Plane

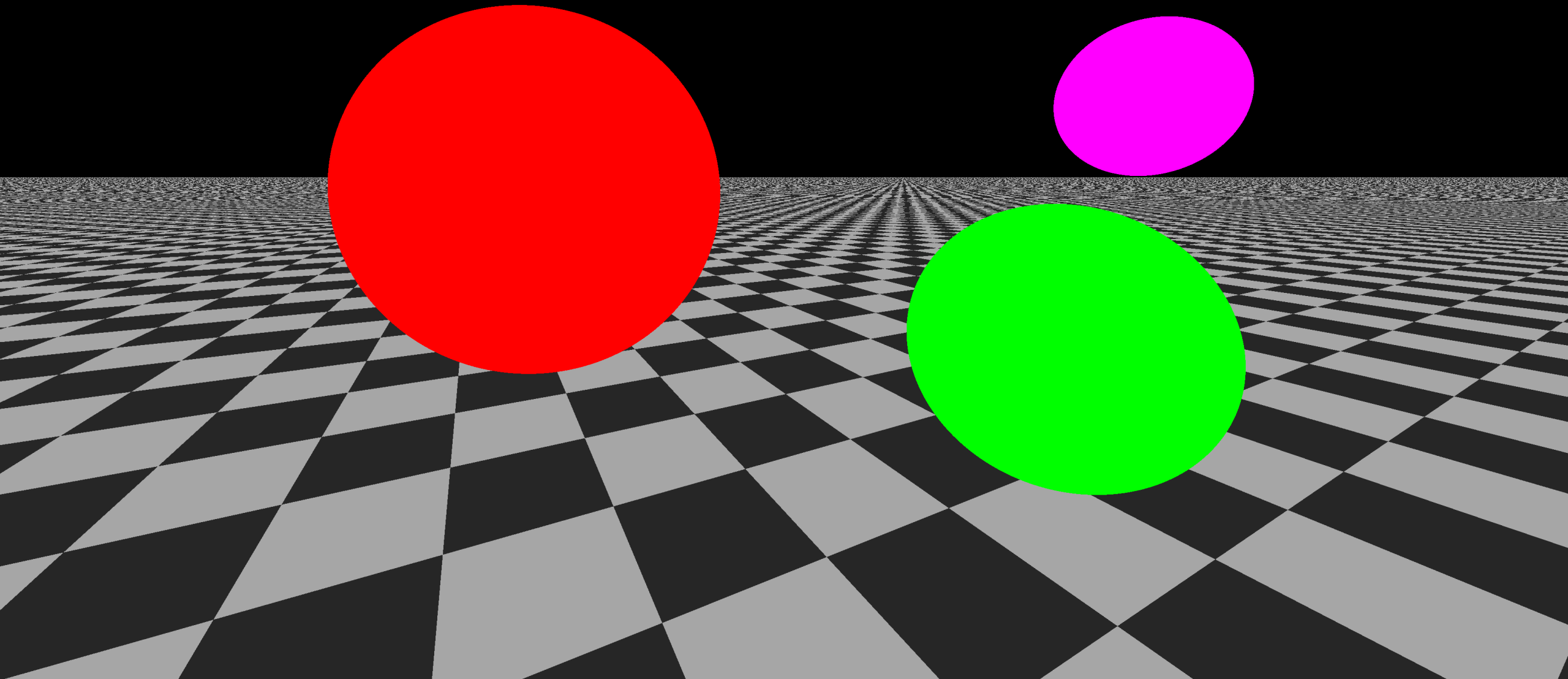
raytracing.fs.glsl

```
float rayPlaneIntersection(vec3 rayStart, vec3 rayDir, vec4 plane, out vec3
      hitColor)
{
  vec3 planeNormal = plane.xyz;
  float d = plane.w;
  float denom = dot(rayDir, planeNormal);
  if (abs(denom) < EPSILON) return INFINITY;
  float t = -(dot(rayStart, planeNormal) - d) / denom;
  if (t > 0.0) {
    hitColor = vec3(0.5, 0.5, 0.5);
    return t;
  }
  else {
    return INFINITY;
  }
}
```

Plane



Checkerboard



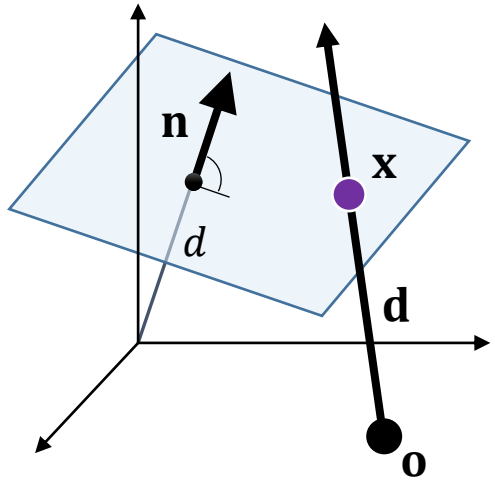
Checkerboard

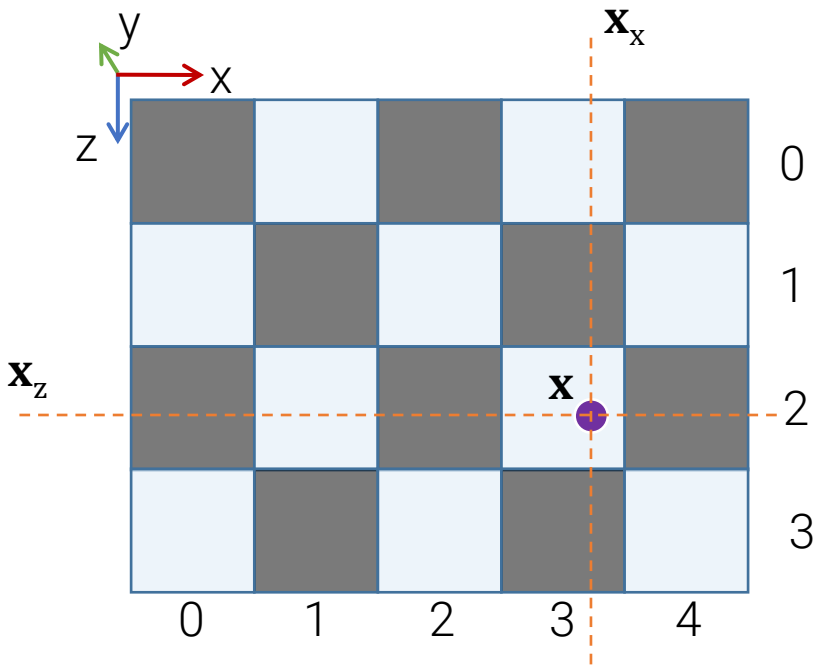
- Compute if in black or white square:
- Calculate hit position $\mathbf{x}$ and use x,z coordinates

$$b = (\lfloor \mathbf{x}_x \rfloor + \lfloor \mathbf{x}_z \rfloor) \% \, 2$$

% is the modulo operation

- $\mathbf{x}$ is in a dark square if $b = 0$





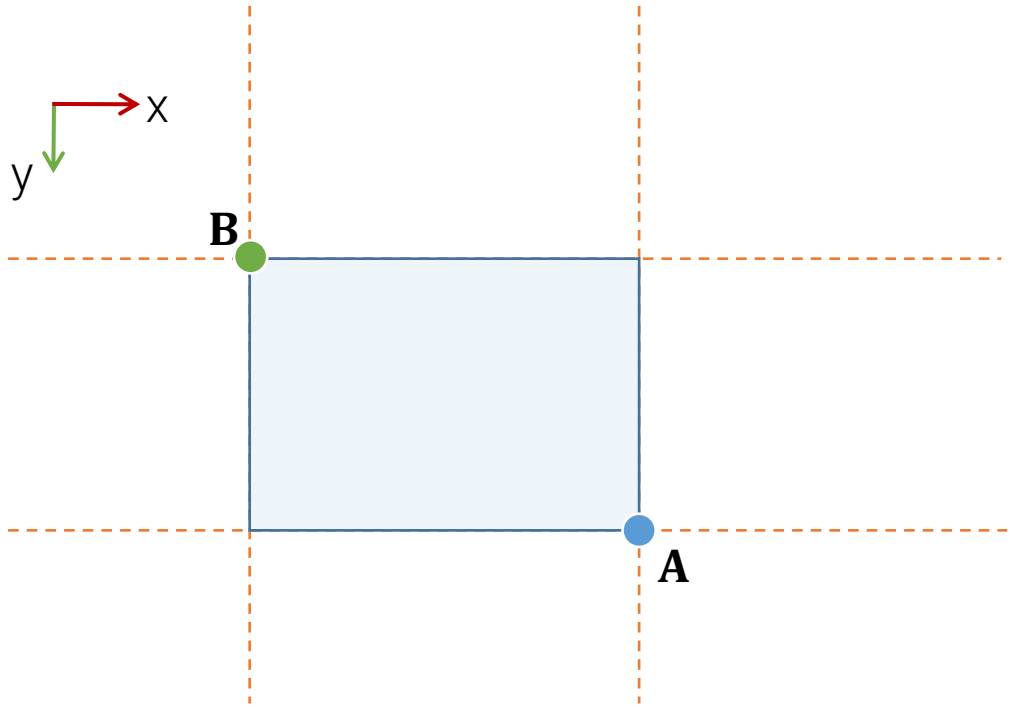
GLSL Code: Checkerboard

raytracing.fs.glsl

```
float rayPlaneIntersection(vec3 rayStart, vec3 rayDir, vec4 plane, out vec3
hitColor)
{
  …
  if (t > 0.0) {
    // checkerboard color
    vec3 hitPos = rayStart + rayDir * t;
    float b = mod(floor(hitPos.z) + floor(hitPos.x), 2.0);
    hitColor = (b + 0.3) * vec3(0.5, 0.5, 0.5);
    return t;
  }
  else {
    return INFINITY;
  }
}
```

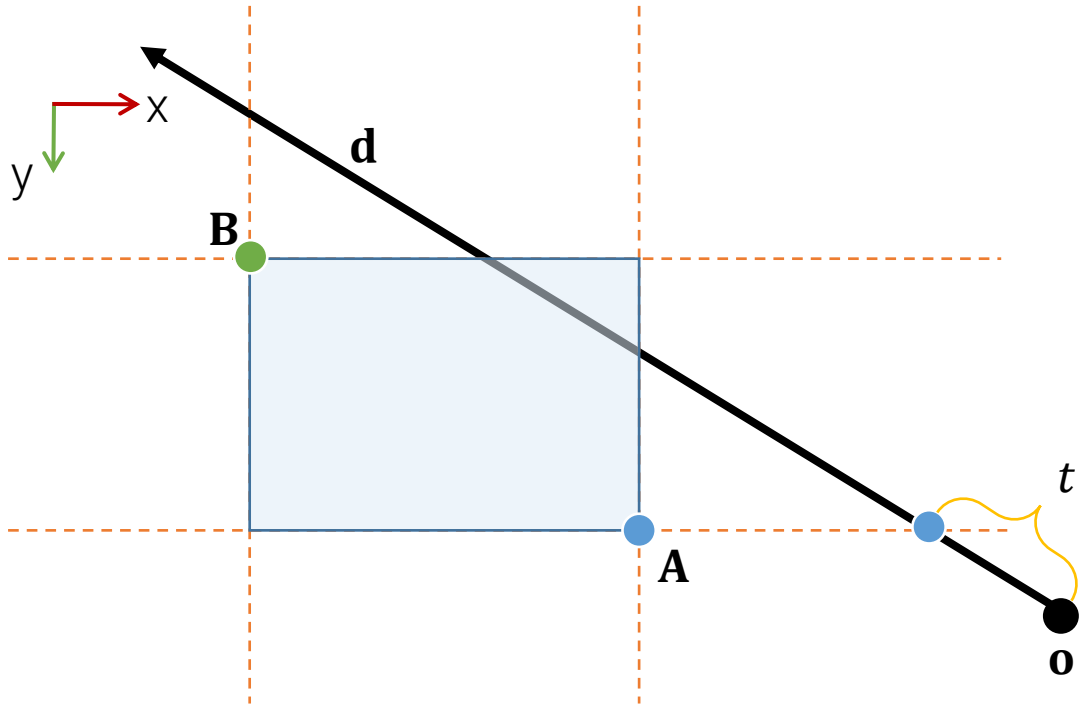
Ray-Cube intersections

- Axis-aligned box defined by 2 points **A** (near) and **B** (far).



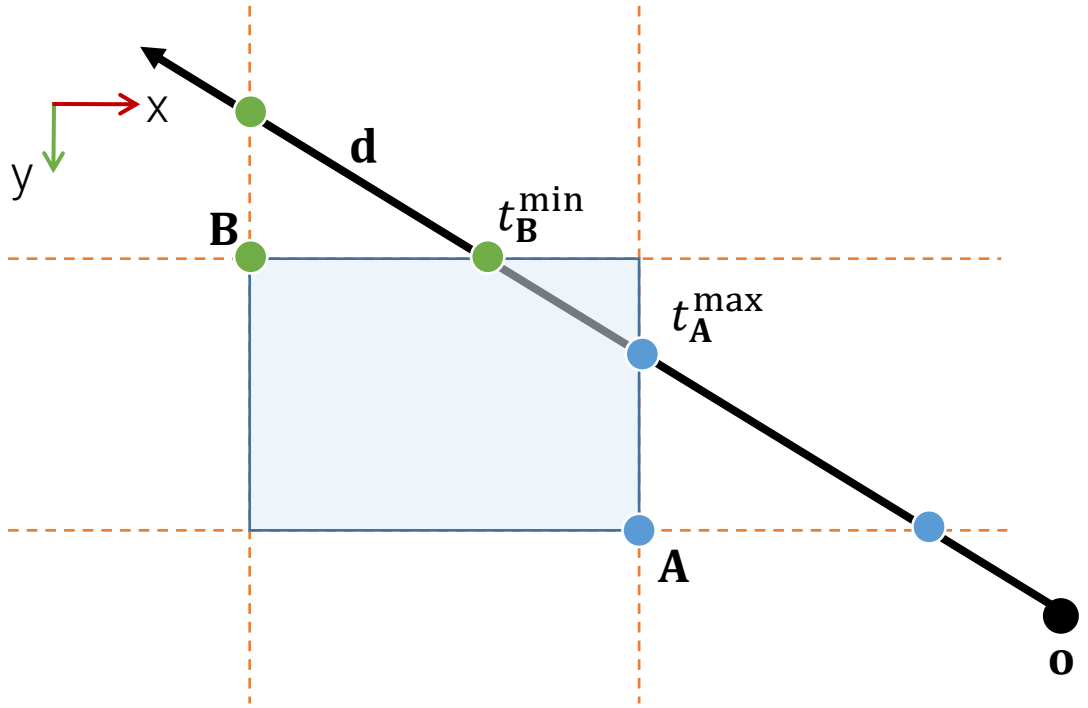
Ray-Cube intersections

- Axis-aligned box defined by 2 points **A** (near) and **B** (far).
- Intersect ray with the planes enclosing the cube (box in 2D).



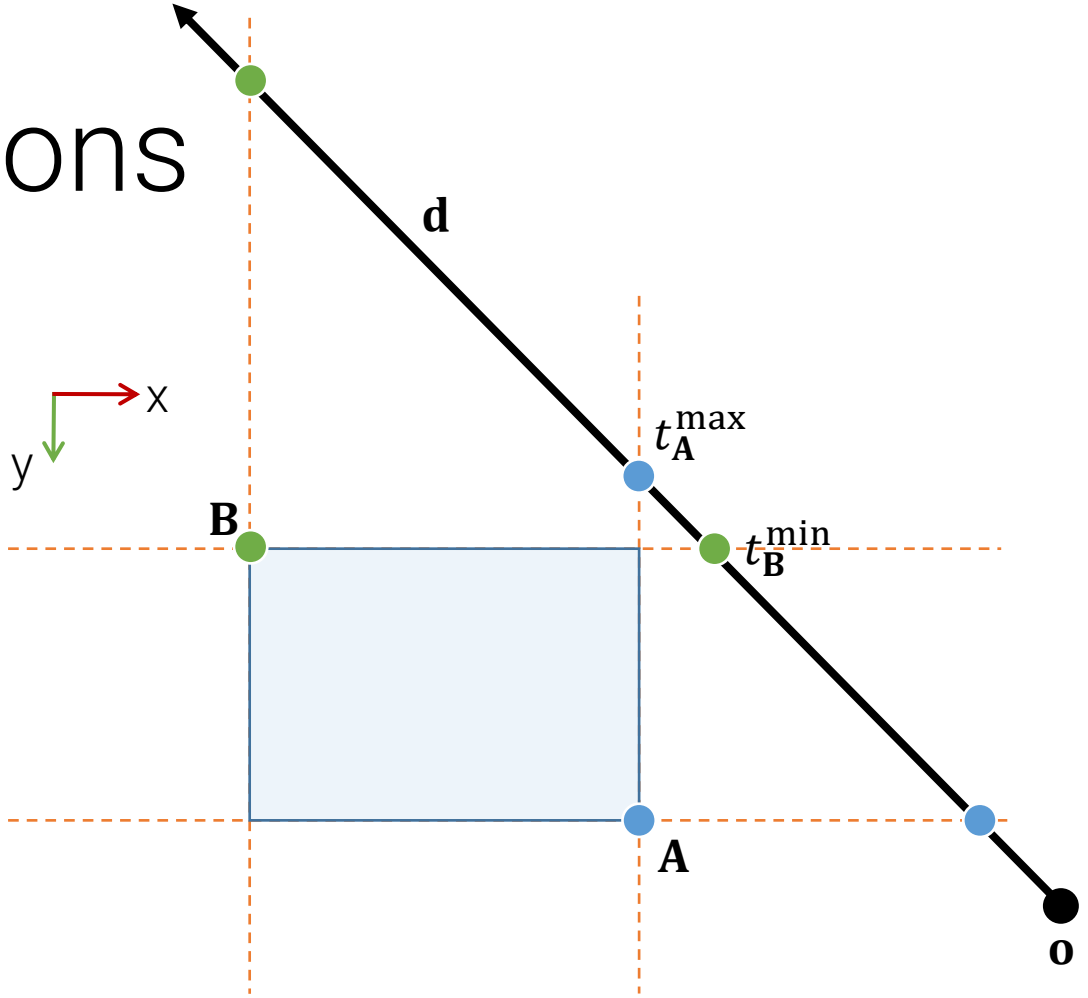
Ray-Cube intersections

- Axis-aligned box defined by 2 points $\mathbf{A}$ (near) and $\mathbf{B}$ (far).
- Intersect ray with the planes enclosing the cube (box in 2D).
- $t_{\mathbf{A}}^{\mathbf{max}}$ is the furthest intersection with near planes (on point $\mathbf{A}$), while $t_{\boldsymbol{B}}^{\mathbf{min}}$ is the closest on any far planes (point $\mathbf{B}$).
- Intersection if $t_{\mathbf{A}}^{\mathbf{max}} \leq t_{\mathbf{B}}^{\mathbf{min}}$.



Ray-Cube intersections

- Axis-aligned box defined by 2 points $\mathbf{A}$ (near) and $\mathbf{B}$ (far).
- Intersect ray with the planes enclosing the cube (box in 2D).
- $t_{\mathbf{A}}^{\mathbf{max}}$ is the furthest intersection with near planes (on point $\mathbf{A}$), while $t_{\boldsymbol{B}}^{\mathbf{min}}$ is the closest on any far planes (point $\mathbf{B}$).
- Intersection if $t_{\mathbf{A}}^{\mathbf{max}} \leq t_{\mathbf{B}}^{\mathbf{min}}$.



GLSL Code: Ray-Cube

raytracing.fs.glsl

```
vec2 intersectCube(vec3 origin, vec3 ray, vec3 cubeMin, vec3 cubeMax) {
  vec3 tMin = (cubeMin - origin) / ray;
  vec3 tMax = (cubeMax - origin) / ray;
  vec3 t1 = min(tMin, tMax);
  vec3 t2 = max(tMin, tMax);
  float tNear = max(max(t1.x, t1.y), t1.z);
  float tFar = min(min(t2.x, t2.y), t2.z);
  return vec2(tNear, tFar);
}
```

GLSL Code: Ray-Cube

raytracing.fs.glsl

```
vec2 intersectCube(vec3 origin, vec3 ray, vec3 cubeMin, vec3 cubeMax) {
  vec3 tMin = (cubeMin - origin) / ray;
  vec3 tMax = (cubeMax - origin) / ray;
```

Ray-plane intersections with axis-aligned planes

GLSL Code: Ray-Cube

raytracing.fs.glsl

```
vec2 intersectCube(vec3 origin, vec3 ray, vec3 cubeMin, vec3 cubeMax) {
  vec3 tMin = (cubeMin - origin) / ray;
  vec3 tMax = (cubeMax - origin) / ray;
  vec3 t1 = min(tMin, tMax);
  vec3 t2 = max(tMin, tMax);
```

Determine near and far planes, but we do not mix x,y,z.

GLSL Code: Ray-Cube

raytracing.fs.glsl

```
vec2 intersectCube(vec3 origin, vec3 ray, vec3 cubeMin, vec3 cubeMax) {
  vec3 tMin = (cubeMin - origin) / ray;
  vec3 tMax = (cubeMax - origin) / ray;
  vec3 t1 = min(tMin, tMax);
  vec3 t2 = max(tMin, tMax);
  float tNear = max(max(t1.x, t1.y), t1.z);
  float tFar = min(min(t2.x, t2.y), t2.z);
  return vec2(tNear, tFar);
}
```

$t_{\mathbf{A}}^{\max}$

$t_{\mathbf{B}}^{\min}$

Compute $t_{\mathbf{A}}^{\max}$ and $t_{\mathbf{B}}^{\min}$ (we mix x,y,z now).

GLSL Code: Ray-Cube

raytracing.fs.glsl

```
vec2 intersectCube(vec3 origin, vec3 ray, vec3 cubeMin, vec3 cubeMax) {
  vec3 tMin = (cubeMin - origin) / ray;
  vec3 tMax = (cubeMax - origin) / ray;
  vec3 t1 = min(tMin, tMax);
  vec3 t2 = max(tMin, tMax);
  float tNear = max(max(t1.x, t1.y), t1.z);
  float tFar = min(min(t2.x, t2.y), t2.z);
  return vec2(tNear, tFar);
}

void addCube( vec3 ro, vec3 rd, vec3 cubeMin, vec3 cubeMax,vec3 color,
       inout float hitDist, inout vec3 hitColor, inout vec3 hitNormal ) {
  vec2 tdist = intersectCube(ro, rd, cubeMax, cubeMin);
  float dist = tdist.x <= tdist.y ? tdist.x : INFINITY;
  dist = dist > 0.0 ? dist : INFINITY;
  if (dist < hitDist) {
    vec3 hit = ro + dist * rd;
    hitNormal = normalForCube(hit, cubeMin, cubeMax);
    …
} }
```

GLSL Code: Ray-Cube

raytracing.fs.glsl

```
vec2 intersectCube(vec3 origin, vec3 ray, vec3 cubeMin, vec3 cubeMax) {
  vec3 tMin = (cubeMin - origin) / ray;
  vec3 tMax = (cubeMax - origin) / ray;
  vec3 t1 = min(tMin, tMax);
  vec3 t2 = max(tMin, tMax);
  float tNear = max(max(t1.x, t1.y), t1.z);
  float tFar = min(min(t2.x, t2.y), t2.z);
  return vec2(tNear, tFar);
}

void addCube( vec3 ro, vec3 rd, vec3 cubeMin, vec3 cubeMax,vec3 color,
      inout float hitDist, inout vec3 hitColor, inout vec3 hitNormal ) {
  vec2 tdist = intersectCube(ro, rd, cubeMax, cubeMin);
  float dist = tdist.x <= tdist.y ? tdist.x : INFINITY;
```

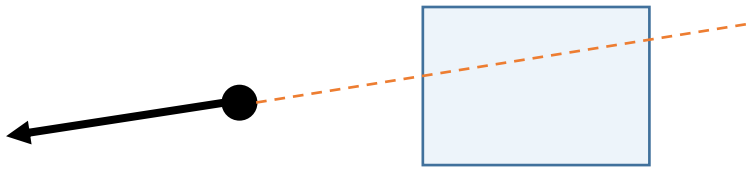
Intersection if $t_{\mathbf{A}}^{\max} \leq t_{\mathbf{B}}^{\min}$

GLSL Code: Ray-Cube

raytracing.fs.glsl

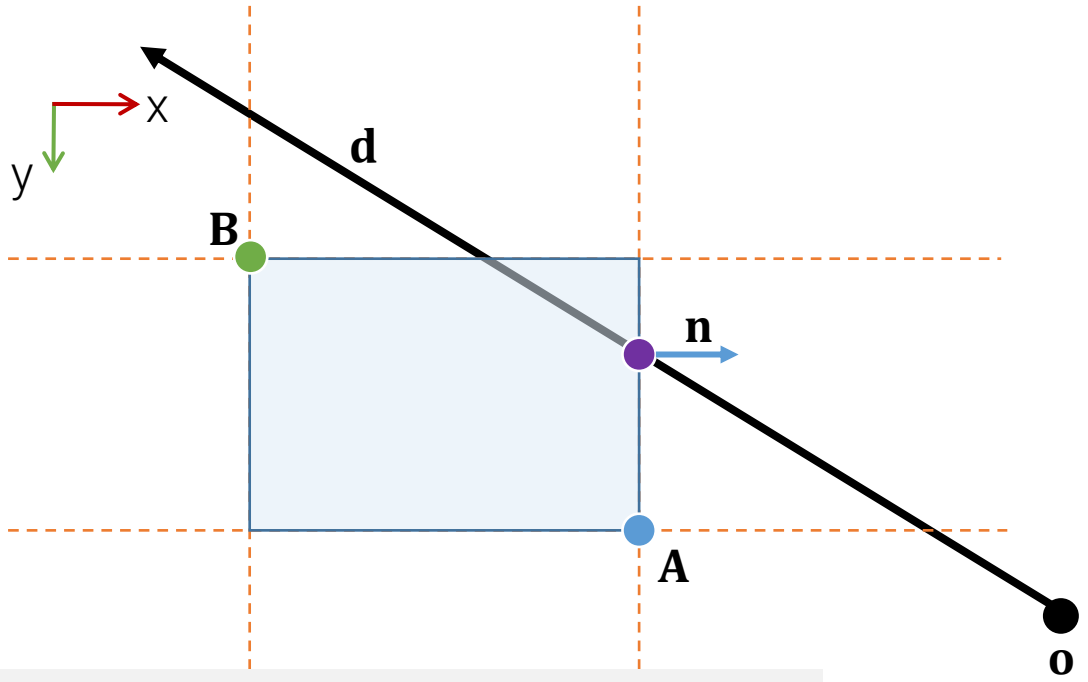
```
vec2 intersectCube(vec3 origin, vec3 ray, vec3 cubeMin, vec3 cubeMax) {
  vec3 tMin = (cubeMin - origin) / ray;
  vec3 tMax = (cubeMax - origin) / ray;
  vec3 t1 = min(tMin, tMax);
  vec3 t2 = max(tMin, tMax);
  float tNear = max(max(t1.x, t1.y), t1.z);    t_A^max
  float tFar = min(min(t2.x, t2.y), t2.z);     t_B^min
  return vec2(tNear, tFar);
}

void addCube( vec3 ro, vec3 rd, vec3 cubeMin, vec3 cubeMax,vec3 color,
      inout float hitDist, inout vec3 hitColor, inout vec3 hitNormal ) {
  vec2 tdist = intersectCube(ro, rd, cubeMax, cubeMin);
  float dist = tdist.x <= tdist.y ? tdist.x : INFINITY;
  dist = dist > 0.0 ? dist : INFINITY;
```

Make sure intersection is in front of origin.



Normal Ray-Cube intersections

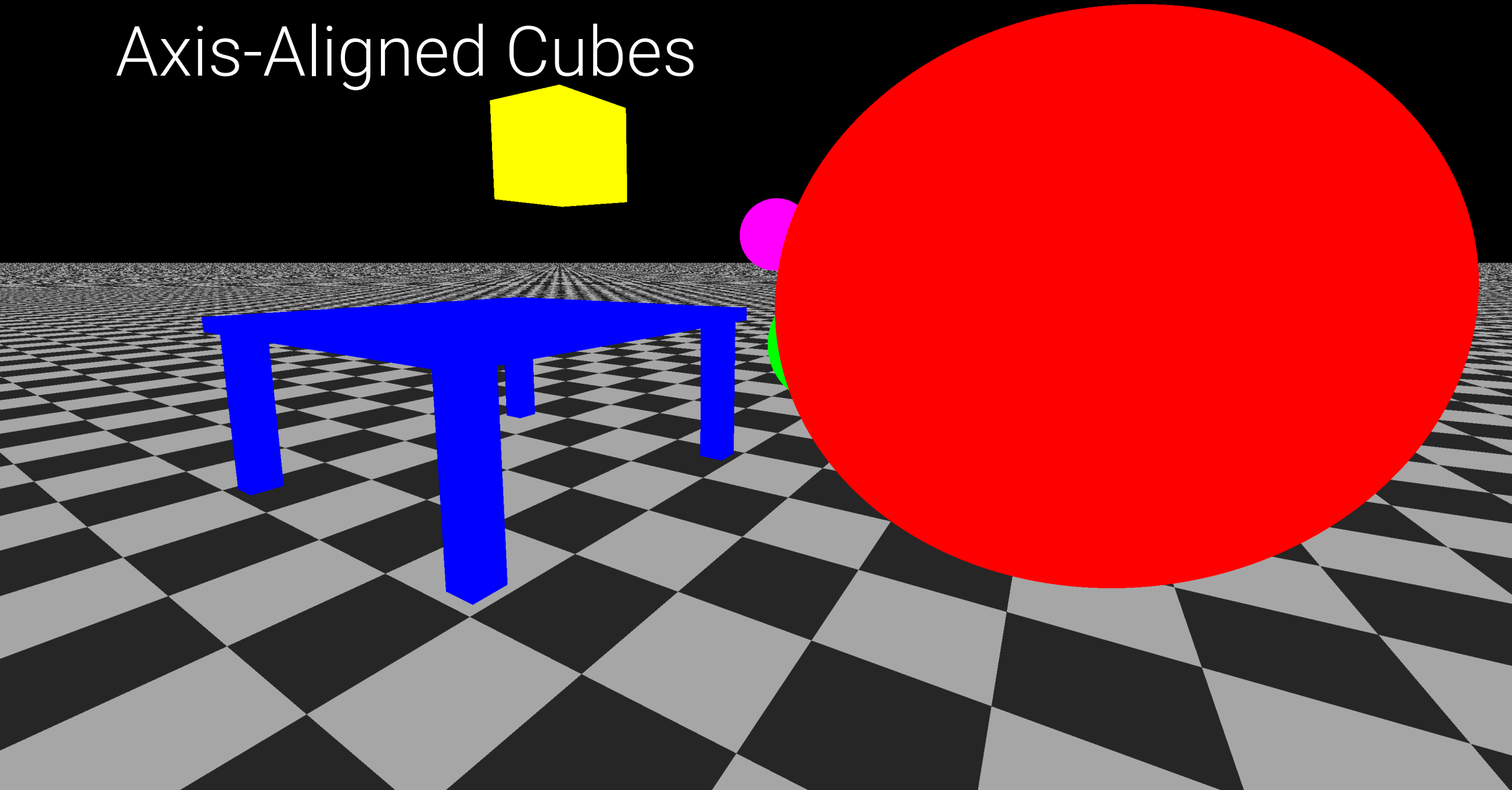
- Get normal close to hit point.
- Remember: axis-aligned



GLSL Code:

```
vec3 normalForCube(vec3 hit, vec3 cubeMin, vec3 cubeMax) {
  if(hit.x < cubeMin.x + RAY_OFFSET) return vec3(-1.0, 0.0, 0.0);
  else if(hit.x > cubeMax.x - RAY_OFFSET ) return vec3(1.0, 0.0, 0.0);
  else if(hit.y < cubeMin.y + RAY_OFFSET ) return vec3(0.0, -1.0, 0.0);
  else if(hit.y > cubeMax.y - RAY_OFFSET ) return vec3(0.0, 1.0, 0.0);
  else if(hit.z < cubeMin.z + RAY_OFFSET ) return vec3(0.0, 0.0, -1.0);
  else return vec3(0.0, 0.0, 1.0);
}
```

Axis-Aligned Cubes

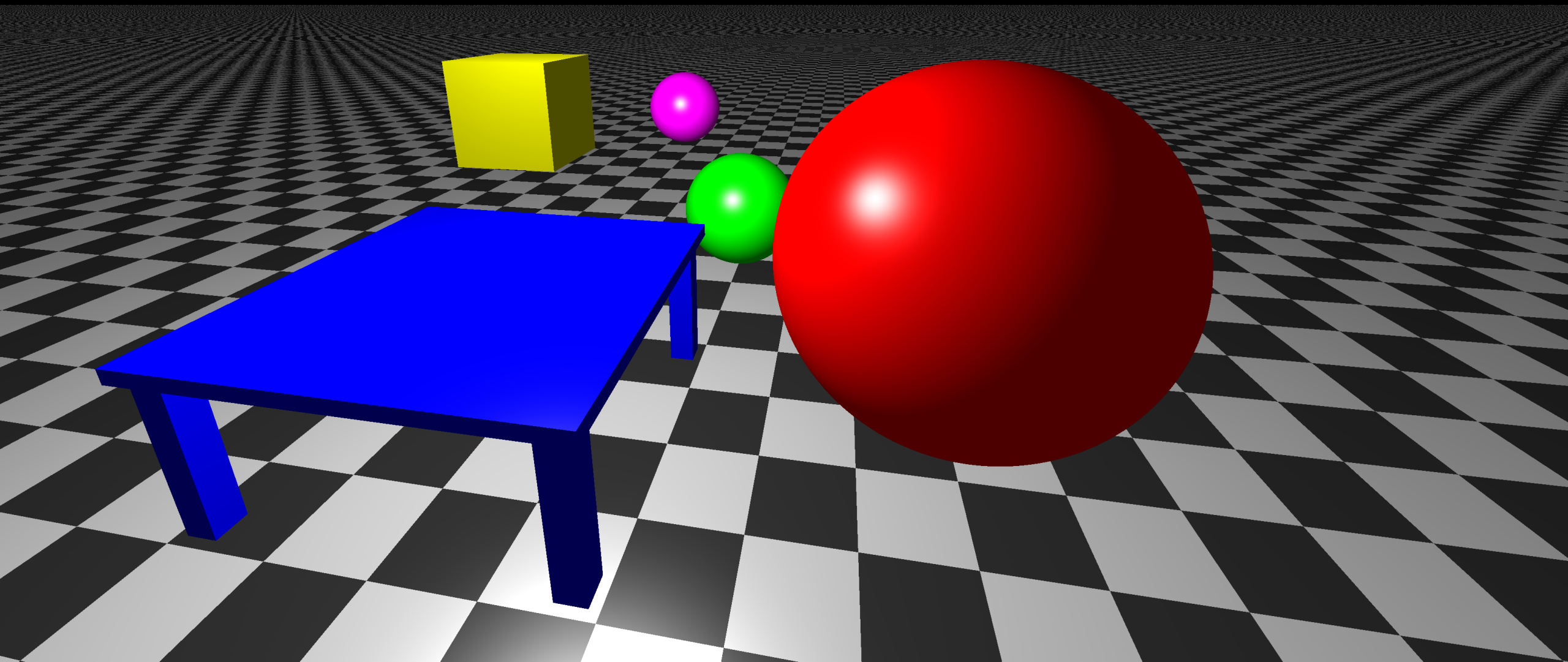


Scene Setup

raytracing.fs.glsl

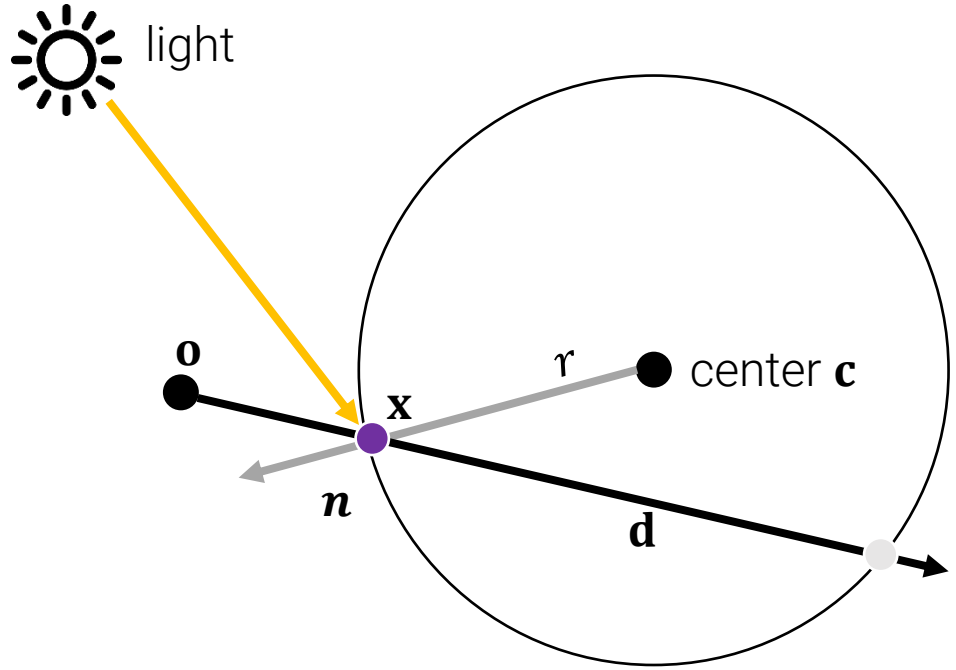
```
float rayTraceScene(vec3 ro /*rayStart*/, vec3 rd /*rayDirection*/, … )
{
    …
    addCube( ro, rd, vec3(0.0, 3.0, 0.0), vec3(1.0, 4.0, 1.0), vec3(1.0, 1.0, 0.0), hitDist,
        hitColor, hitNormal  ); // add floating cube

    // table top
    addCube(ro, rd, vec3(-2, 1.65, -2), vec3(2, 1.8, 2), vec3(0.0, 0.0, 1.0), hitDist,
        hitColor, hitNormal  );
    // table legs
    addCube(ro, rd, vec3(-1.9, 0, -1.9), vec3(-1.6, 1.65, -1.6), vec3(0.0, 0.0, 1.0),
        hitDist, hitColor, hitNormal  );
    addCube(ro, rd, vec3(-1.9, 0, 1.6), vec3(-1.6, 1.65, 1.9), vec3(0.0, 0.0, 1.0), hitDist,
        hitColor, hitNormal  );
    addCube(ro, rd, vec3(1.6, 0, 1.6), vec3(1.9, 1.65, 1.9), vec3(0.0, 0.0, 1.0), hitDist,
        hitColor, hitNormal  );
    addCube(ro, rd, vec3(1.6, 0, -1.9), vec3(1.9, 1.65, -1.6), vec3(0.0, 0.0, 1.0), hitDist,
        hitColor, hitNormal  );
    …
}
```

Shading



Local Shading

- Compute Blinn-Phong shading model at the intersection point (hit point).
- We need:
  - a surface point
  - the normal
  - colors
  - light position

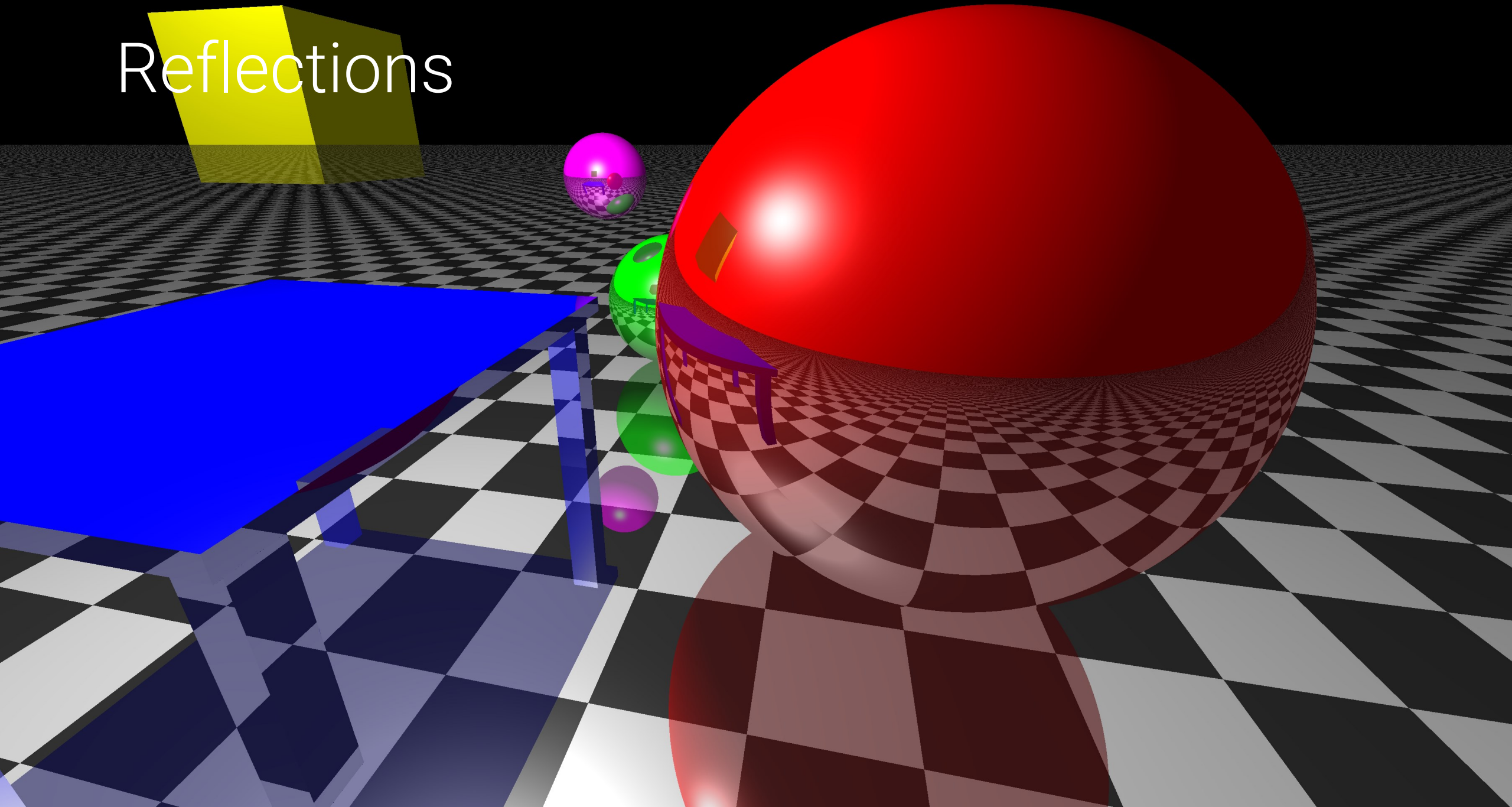


GLSL Code: Shading

raytracing.fs.glsl

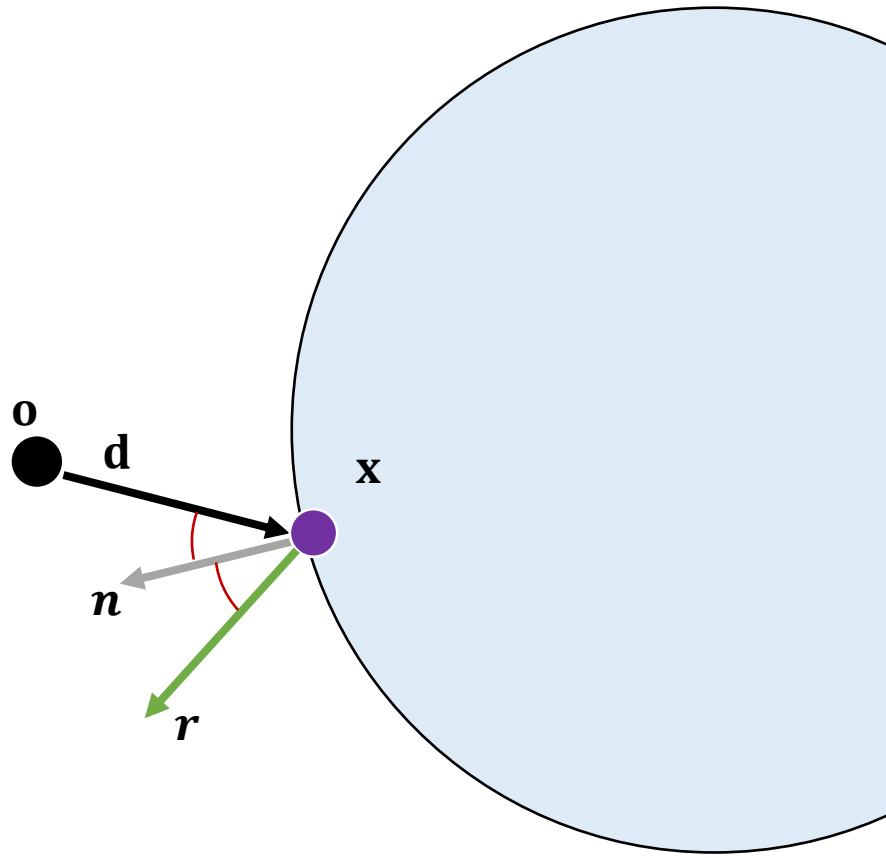
```
vec3 calcLighting(vec3 hitPoint, vec3 normal, vec3 inRay, vec3 color) {
  vec3 ambient = vec3(0.3, 0.3, 0.3);
  vec3 lightVec = lightPosition - hitPoint;
  vec3 lightDir = normalize(lightVec);
  float diff = max(dot(normal, lightDir),0.0);
  vec3 h = normalize(-inRay + lightDir); // half vector
  float ndoth = max(dot(normal, h),0.0);
  float spec = max(pow(ndoth, 50.0),0.0);
  return min((ambient + vec3(diff)) * color + vec3(spec), 1.0);
}

void main() {
  …
  float dist = rayTraceScene(rayStart, rayDirection, hitNormal, hitColor);
  vec3 hitPoint = rayStart + rayDirection * dist;
  gl_FragColor.rgb = calcLighting(hitPoint, hitNormal, rayDirection, hitColor);
}
```

Reflections



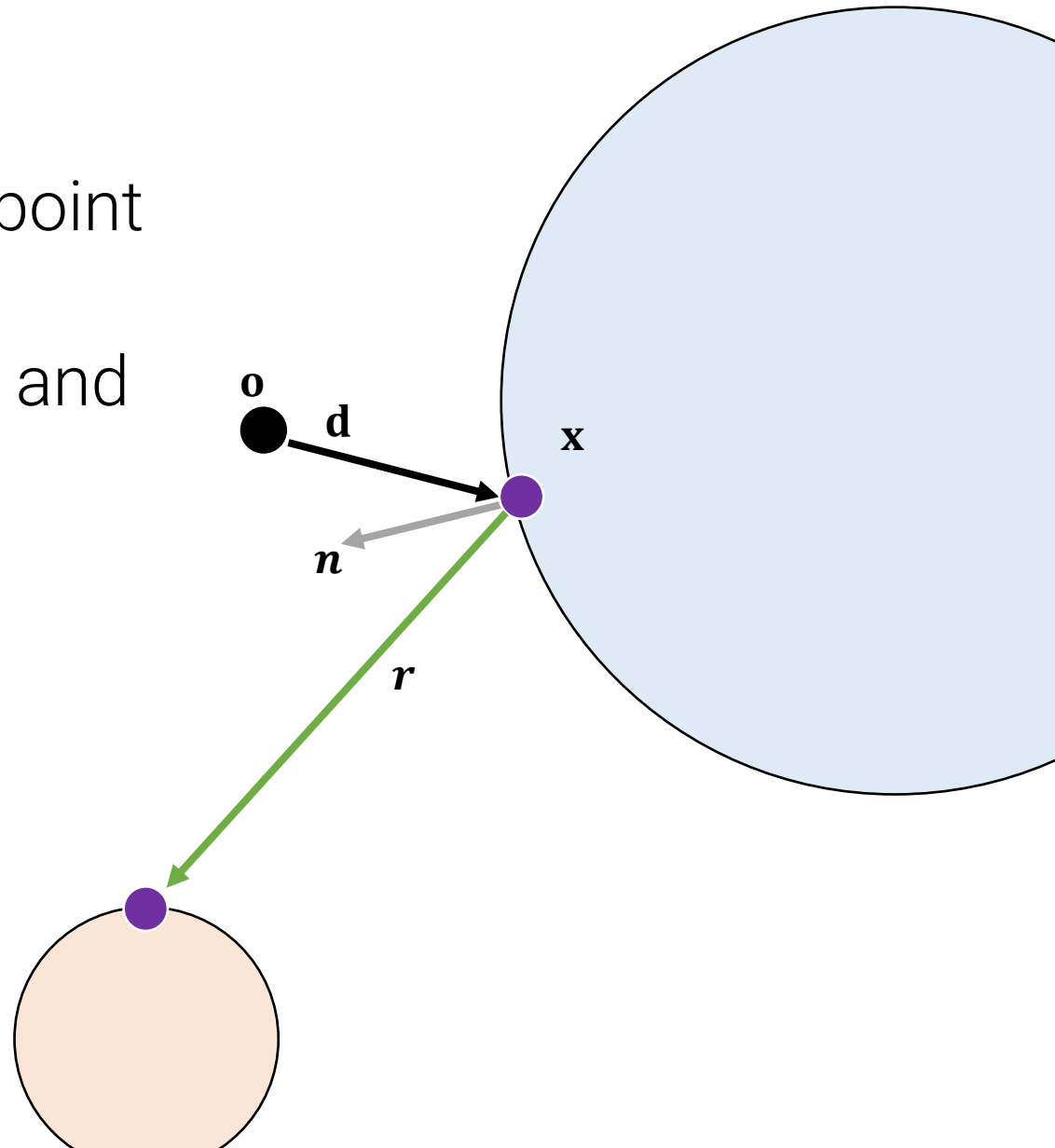
Reflections

- Reflect the incoming ray $\mathbf{d}$ at the hit point $\mathbf{x}$ with normal $\mathbf{n}$.



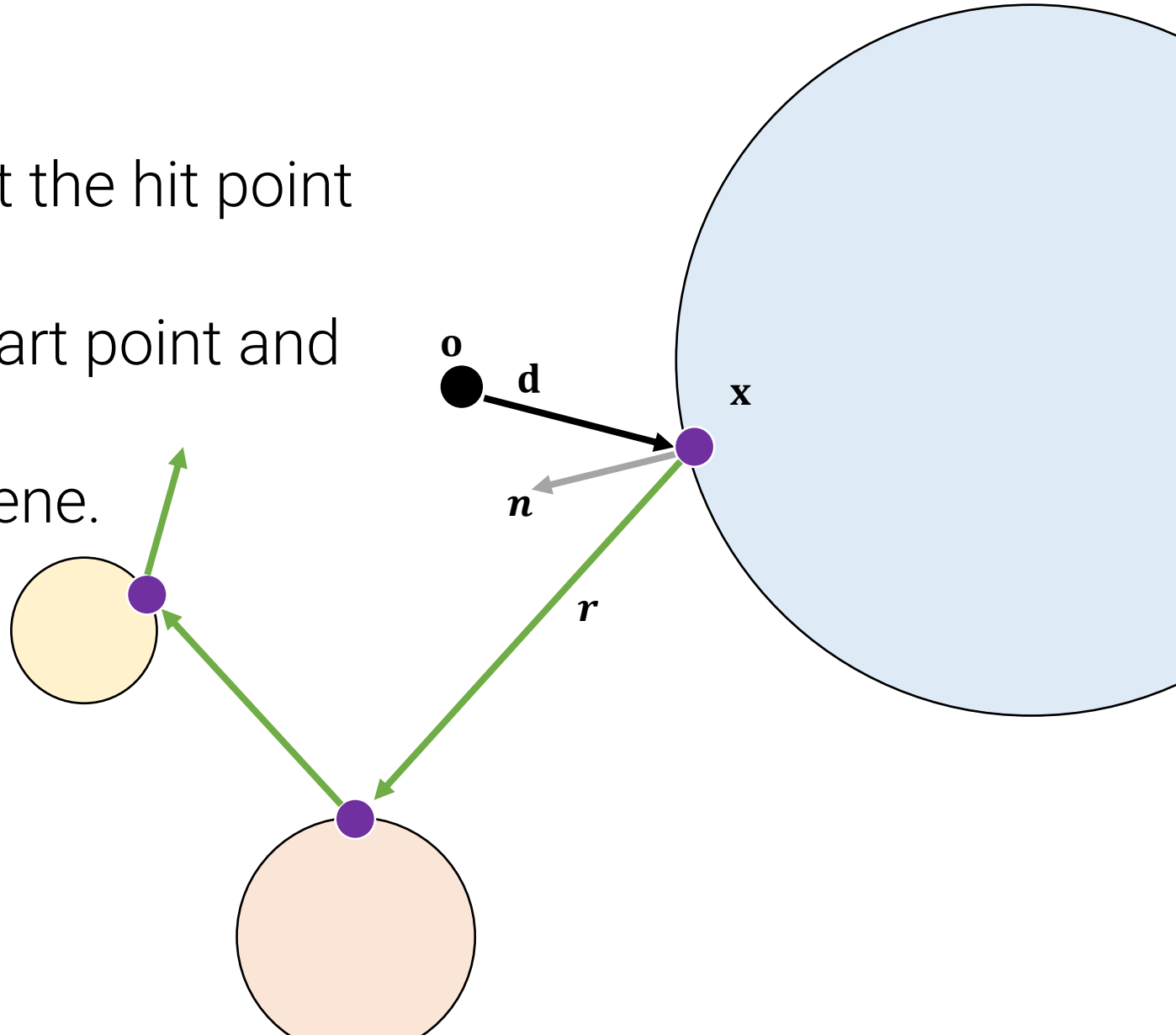
Reflections

- Reflect the incoming ray $\mathbf{d}$ at the hit point $\mathbf{x}$ with normal $\mathbf{n}$.
- Shoot a new ray with $\mathbf{x}$ as start point and $\mathbf{r}$ as the direction.
- Intersect (again) with the scene.



Reflections

- Reflect the incoming ray $\mathbf{d}$ at the hit point $\mathbf{x}$ with normal $\mathbf{n}$.
- Shoot a new ray with $\mathbf{x}$ as start point and $\mathbf{r}$ as the direction.
- Intersect (again) with the scene.
- Continue iteratively …



GLSL Code: Reflections

raytracing.fs.glsl

```
void main() {
   …
   vec3 color = vec3( 0.0 ); float hits = 0.0;

   for (int i = 0; i < maxDepth; i++)
   {
      float dist = rayTraceScene(rayStart, rayDirection, hitNormal, hitColor);
      if (dist >= INFINITY) { break; }
      hits += 1.0;
      vec3 nearestHit = rayStart + dist * rayDirection;
      color.rgb += calcLighting(nearestHit, hitNormal, rayDirection, hitColor);

      rayDirection = reflect(rayDirection, hitNormal);
      rayDirection = normalize(rayDirection);
      rayStart = nearestHit + hitNormal * RAY_OFFSET;
   }

   if (hits > 0.0) color /= hits;
   gl_FragColor.rgb = vec3(color);
}
```

GLSL Code: Reflections

raytracing.fs.glsl

```
void main() {
    …
    for (int i = 0; i < maxDepth; i++)
    {
```

Maximum number of reflection rays (bounces).

GLSL Code: Reflections

raytracing.fs.glsl

```
void main() {
    …
    for (int i = 0; i < maxDepth; i++)
    {
        float dist = rayTraceScene(rayStart, rayDirection, hitNormal, hitColor);
        if (dist >= INFINITY) { break; }
        hits += 1.0;
```

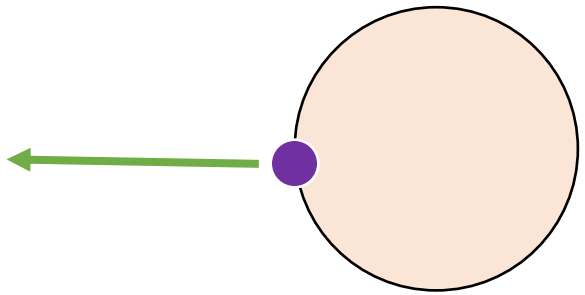
Stop loop if we did not hit any object (ray is leaving our scene).

GLSL Code: Reflections

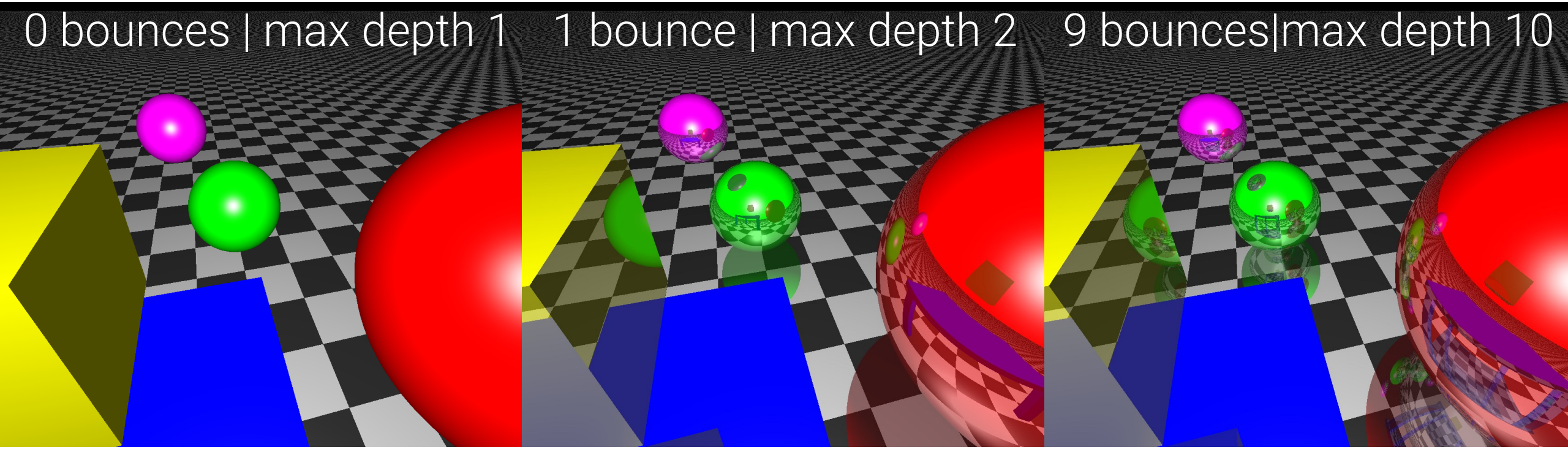
raytracing.fs.glsl

```
void main() {
    …
    for (int i = 0; i < maxDepth; i++)
    {
        float dist = rayTraceScene(rayStart, rayDirection, hitNormal, hitColor);
        if (dist >= INFINITY) { break; }
        hits += 1.0;
        vec3 nearestHit = rayStart + dist * rayDirection;
        color.rgb += calcLighting(nearestHit, hitNormal, rayDirection, hitColor);

        rayDirection = reflect(rayDirection, hitNormal);
        rayDirection = normalize(rayDirection);
        rayStart = nearestHit + hitNormal * RAY_OFFSET;
```

Compute reflected ray direction and starting point.
The starting point is offset by a tiny fraction to avoid precision issues.



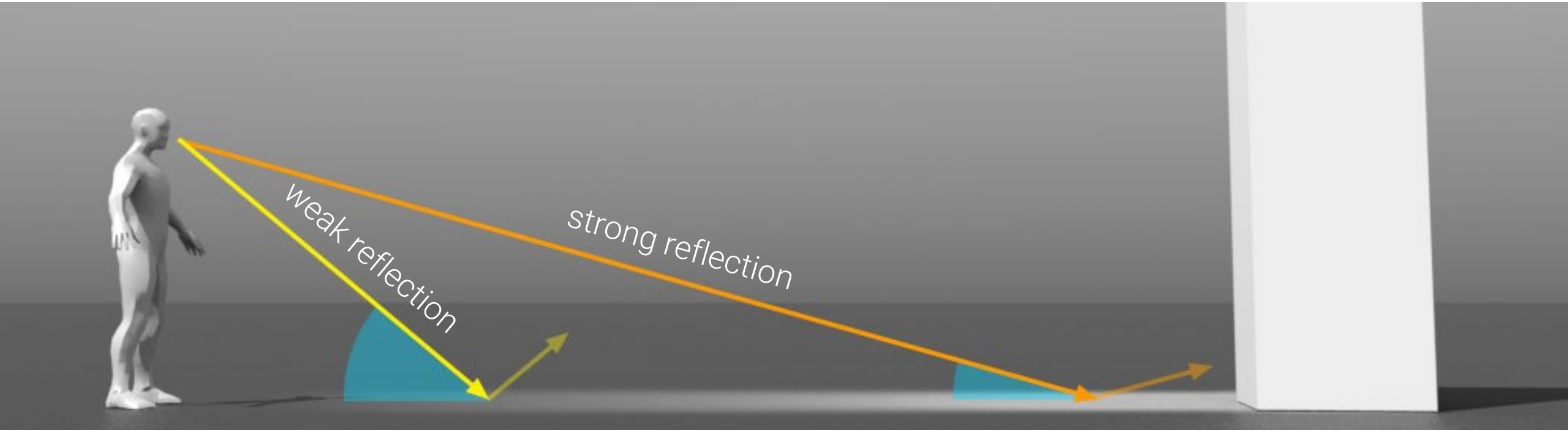
Number of Bounces



With more bounces we get reflections in the reflections.

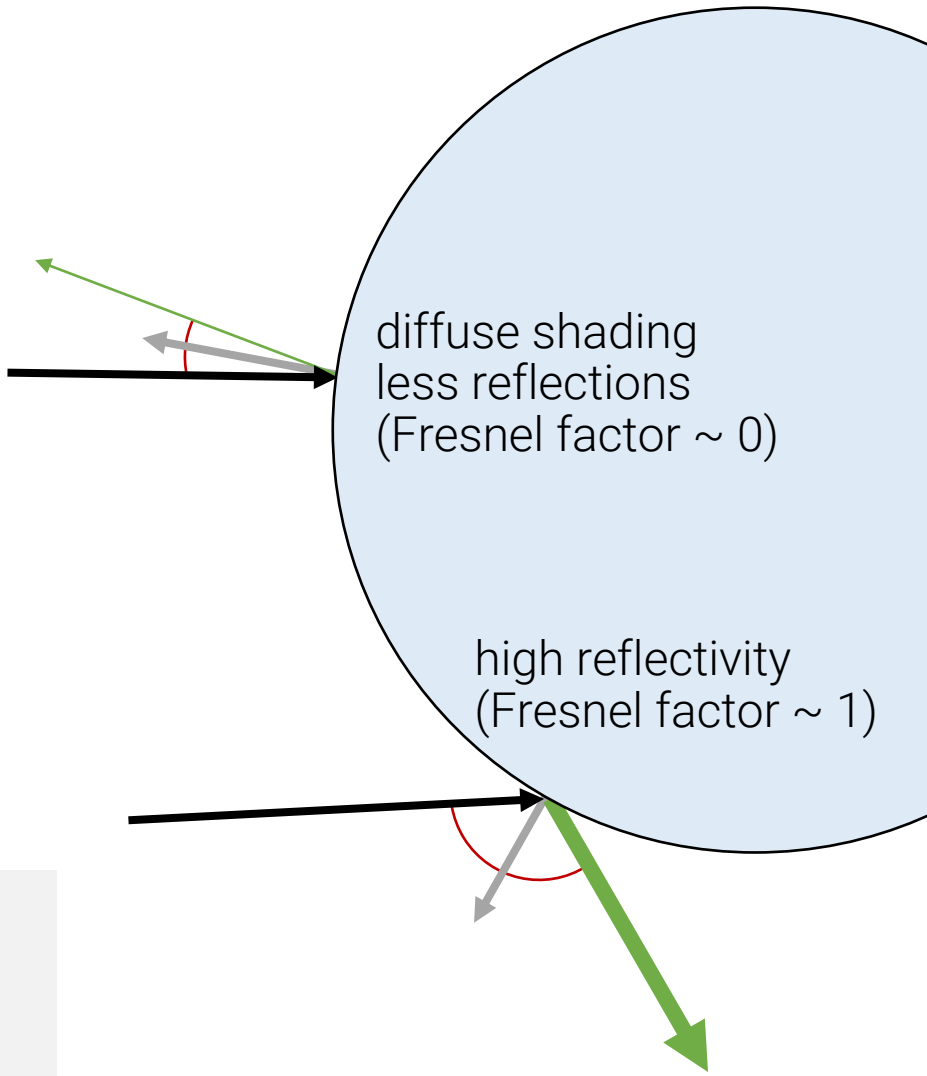
Recap: Fresnel [Freh-´nell]

Augustin-Jean Fresnel



Fresnel Reflections

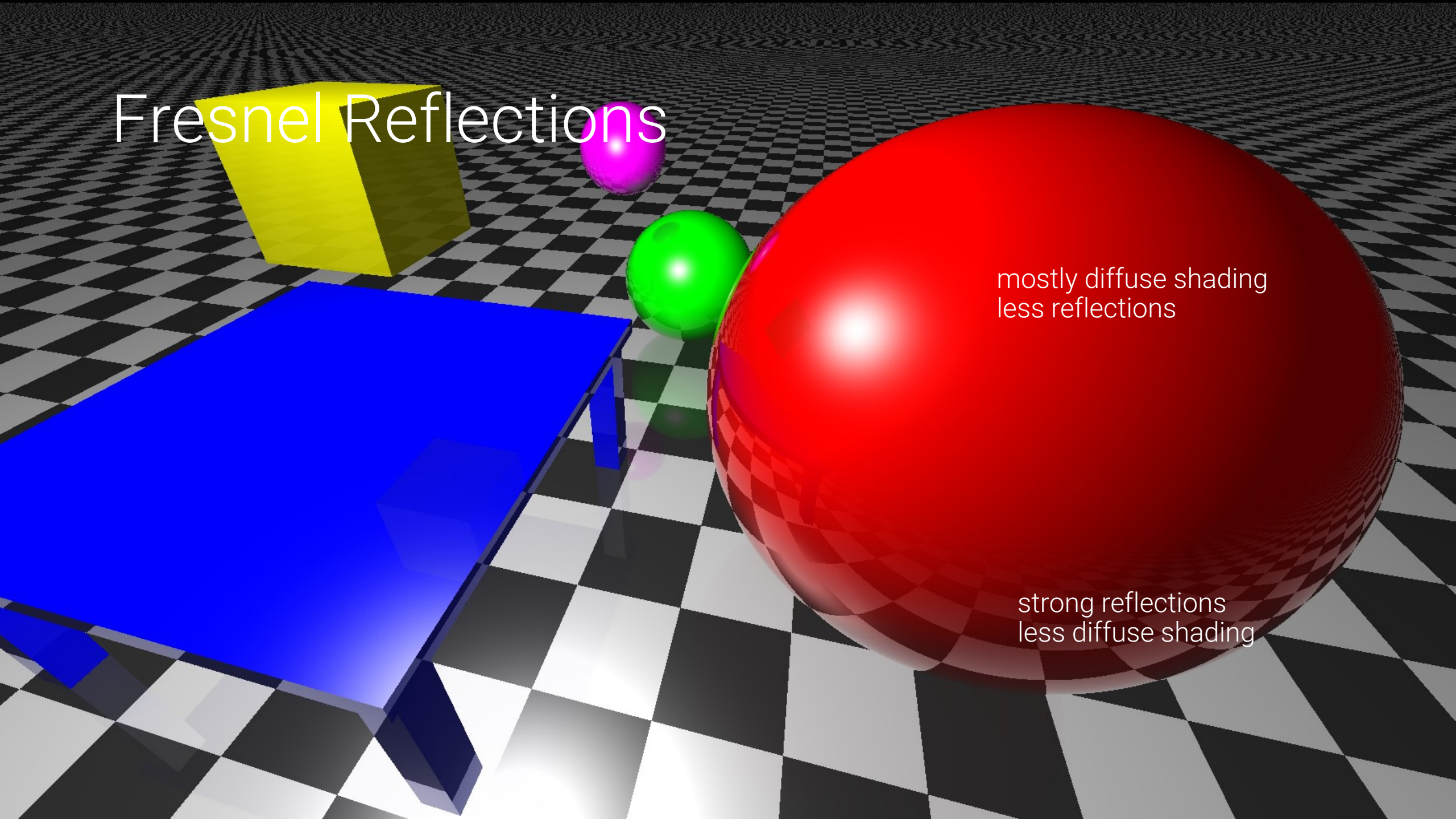
- Compute fresnel factor based on the angle between the normal and the ray direction.



GLSL Code (not from PBR):

```
float calcFresnel(vec3 normal, vec3 inRay) {
  float bias = 0.0;
  float cosTheta = clamp(dot(normal, -inRay), 0.0, 1.0);
  return clamp(bias + pow(1.0 - cosTheta, 2.0), 0.0, 1.0);
}
```

Fresnel Reflections



GLSL Code: Fresnel Reflections

raytracing.fs.glsl

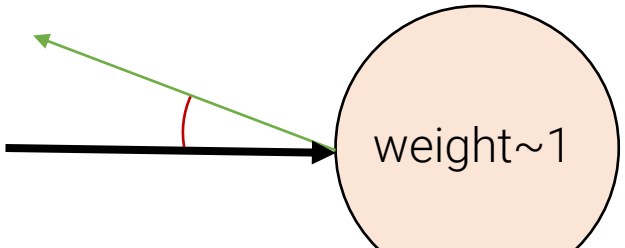
```
void main() {
 …
 for (int i = 0; i < maxDepth; i++)
 {
   float dist = rayTraceScene(rayStart, rayDirection, hitNormal, hitColor);
   if (dist >= INFINITY) {break;}

   float fresnel = calcFresnel(hitNormal, rayDirection);
   float weight = (1.0-fresnel)*(1.0-hits);
   hits += weight;
   vec3 nearestHit = rayStart + dist * rayDirection;
   color.rgb += calcLighting(nearestHit, hitNormal, rayDirection, hitColor) * weight;

   rayDirection = reflect(rayDirection, hitNormal);
   rayDirection = normalize(rayDirection);
   rayStart = nearestHit + hitNormal * RAY_OFFSET;
 }
 …
}
```

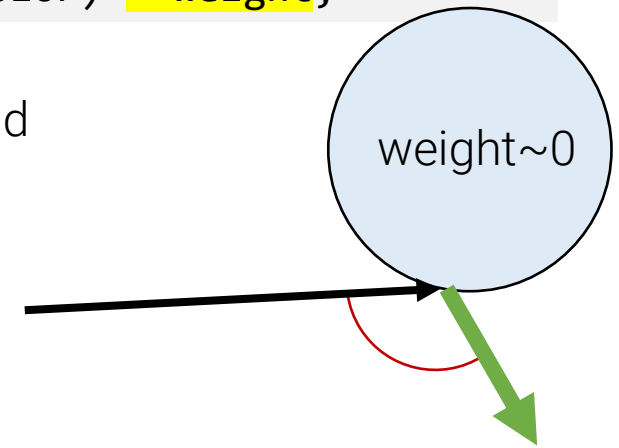
GLSL Code: Fresnel Reflections

raytracing.fs.glsl

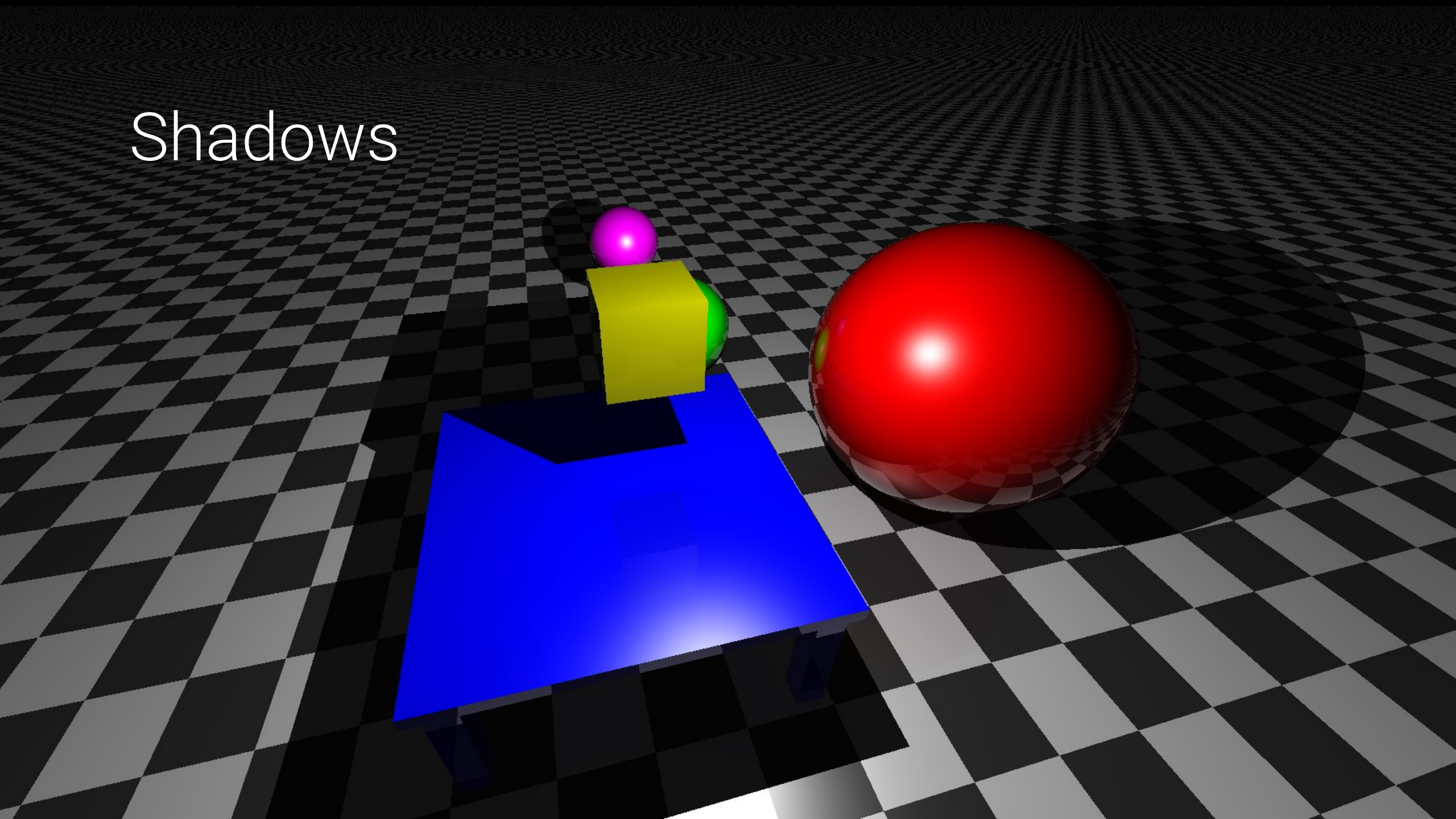
```
void main() {
…
for (int i = 0; i < maxDepth; i++)
{
    float dist = rayTraceScene(rayStart, rayDirection, hitNormal, hitColor);
    if (dist >= INFINITY) {break;}

    float fresnel = calcFresnel(hitNormal, rayDirection);
    float weight = (1.0-fresnel)*(1.0-hits);
    hits += weight;
    vec3 nearestHit = rayStart + dist * rayDirection;
    color.rgb += calcLighting(nearestHit, hitNormal, rayDirection, hitColor) * weight;
```

Compute Fresnel factor and a weight (Note: hits is initially 0).
High Fresnel factor and/or high `hits` (previous weights) leads to low weights and thus, to less contribution from the local shading in the final color.



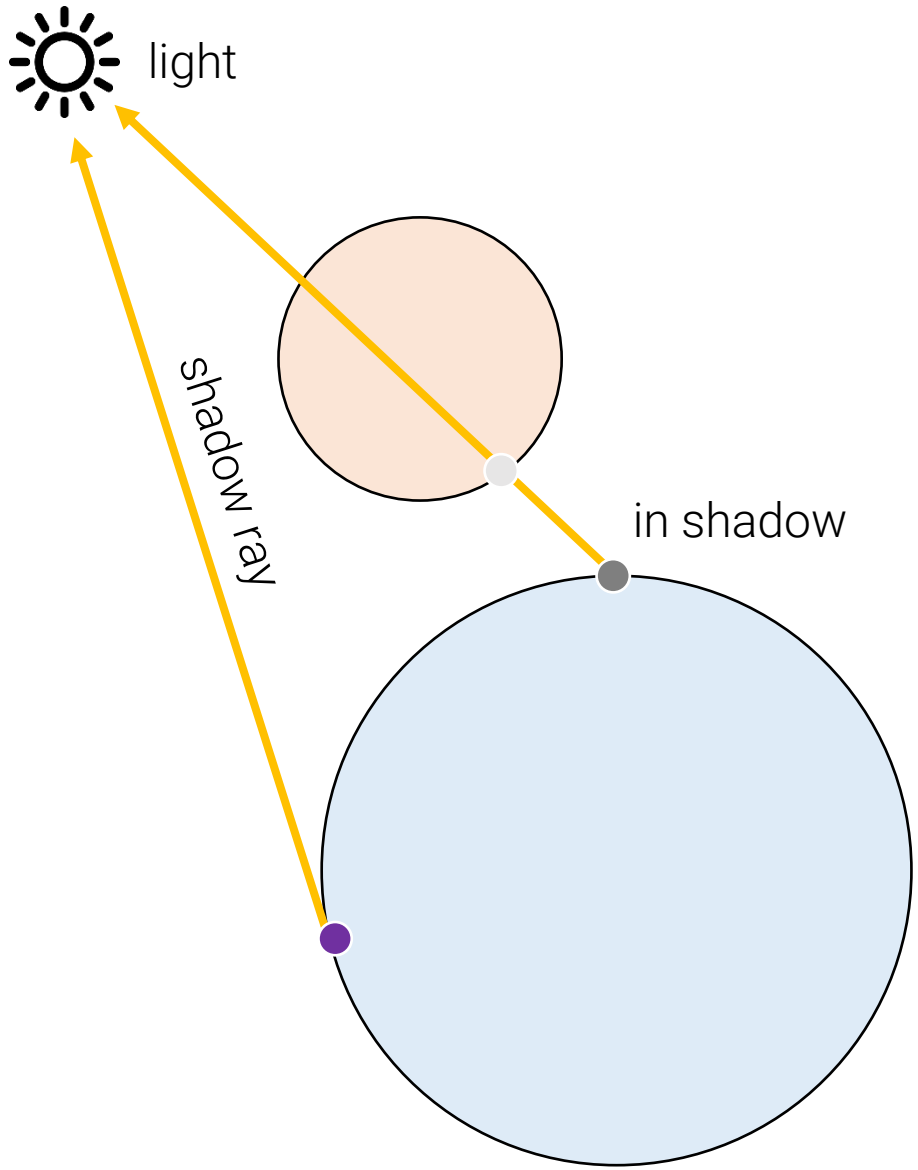


Shadows



Ray tracing shadows

- For a hit point: cast a ray towards the light (shadow ray)
- Check for intersections in-between object and light source (compare distances)
- Evaluate local illumination model (diffuse and specular) only if path to light is not blocked.



GLSL Code: Shadows

raytracing.fs.glsl

```
vec3 calcLighting(vec3 hitPoint, vec3 normal, vec3 inRay, vec3 color) {
   …
   float lightDist = length( lightVec );
   vec3 hitNormal, hitColor, shading=vec3(0.0);
   float shadowRayDist = rayTraceScene(hitPoint + lightDir*RAY_OFFSET, lightDir, hitNormal,
     hitColor );
   if(shadowRayDist < lightDist) {
     shading += ambient * color;
   } else {
     float diff = max(dot(normal, lightDir),0.0);
     vec3 h = normalize(-inRay + lightDir);
     float ndoth = max(dot(normal, h),0.0);
     float spec = max(pow(ndoth, 50.0),0.0);
     shading += min((ambient + vec3(diff)) * color + vec3(spec), 1.0);
   }
   return shading;
}
```

GLSL Code: Shadows

raytracing.fs.glsl

```
vec3 calcLighting(vec3 hitPoint, vec3 normal, vec3 inRay, vec3 color) {
   vec3 lightVec = lightPosition - hitPoint;
   vec3 lightDir = normalize(lightVec);
   float lightDist = length(lightVec);
   vec3 hitNormal, hitColor, shading=vec3(0.0);
   float shadowRayDist = rayTraceScene(hitPoint + lightDir*RAY_OFFSET, lightDir, hitNormal,
     hitColor );
   if(shadowRayDist < lightDist) {
     shading += ambient * color;
```

Cast a ray towards the light (shadow ray). The starting point is offset again to avoid any precision issues.

Compare distances of the light and the shadow-ray intersection.

GLSL Code: Shadows

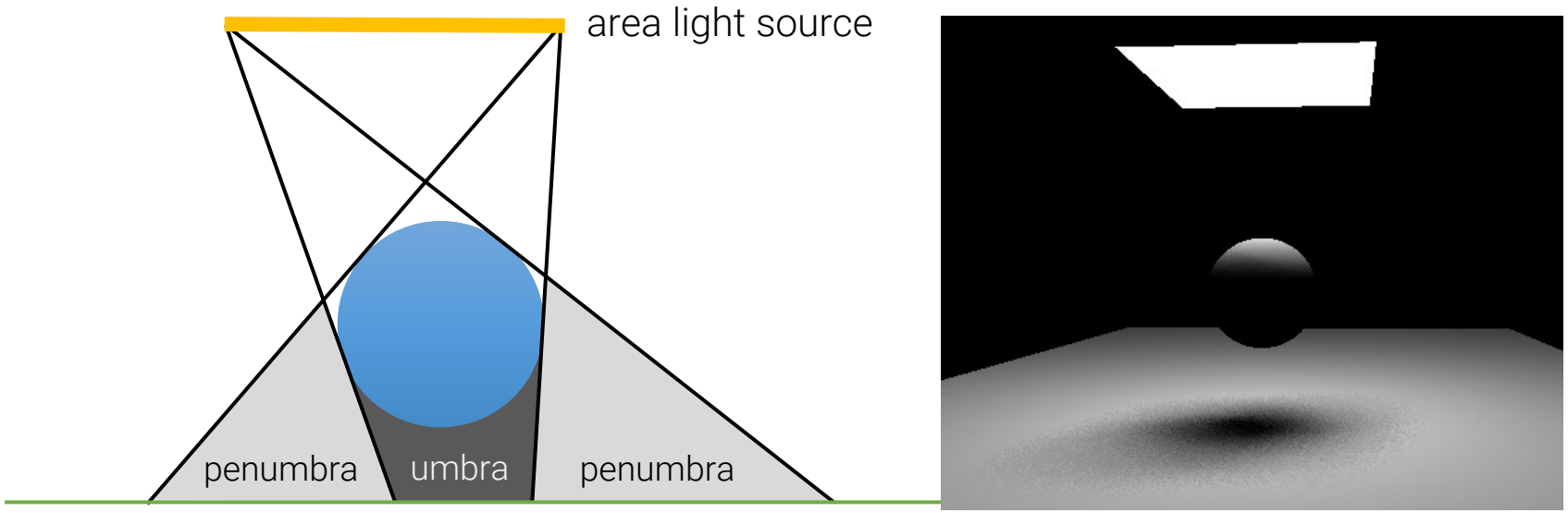
raytracing.fs.glsl

```
vec3 calcLighting(vec3 hitPoint, vec3 normal, vec3 inRay, vec3 color) {
   vec3 lightVec = lightPosition - hitPoint;
   vec3 lightDir = normalize(lightVec);
   float lightDist = length(lightVec);
   vec3 hitNormal, hitColor, shading=vec3(0.0);
   float shadowRayDist = rayTraceScene(hitPoint + lightDir*RAY_OFFSET, lightDir, hitNormal,
     hitColor );
   if(shadowRayDist < lightDist) {
     shading += ambient * color;
   } else {
     float diff = max(dot(normal, lightDir),0.0);
     vec3 h = normalize(-inRay + lightDir);
     float ndoth = max(dot(normal, h),0.0);
     float spec = max(pow(ndoth, 50.0),0.0);
     shading += min((ambient + vec3(diff)) * color + vec3(spec), 1.0);
   }
```

If in shadow, only compute ambient term.
Otherwise, evaluate local shading model (diffuse + specular).

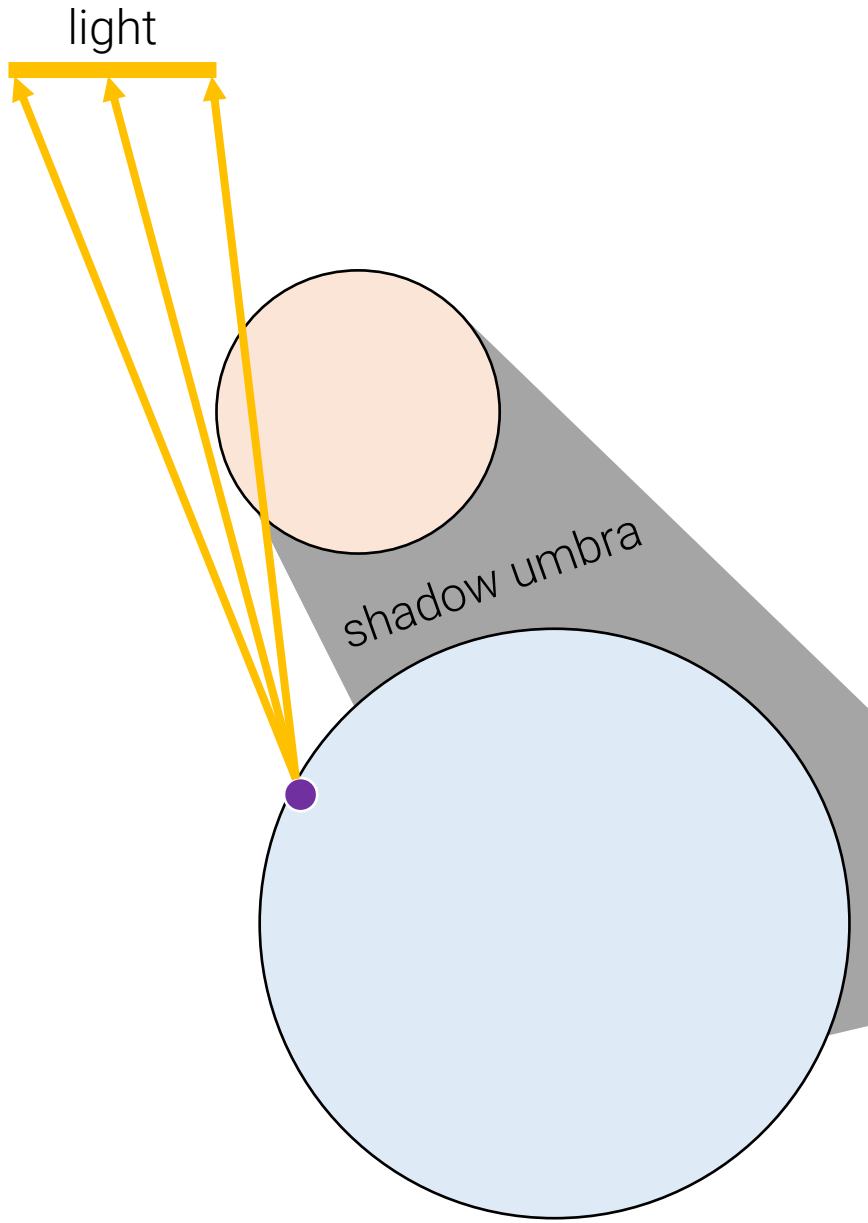
Ray tracing soft shadows

- Soft-shadows (Penumbras) are created by area lights.



Ray tracing soft shadows

- Approximation:
  - Distribute multiple shadow rays across the region
  - Weight shadow contributions (similar to PCF)

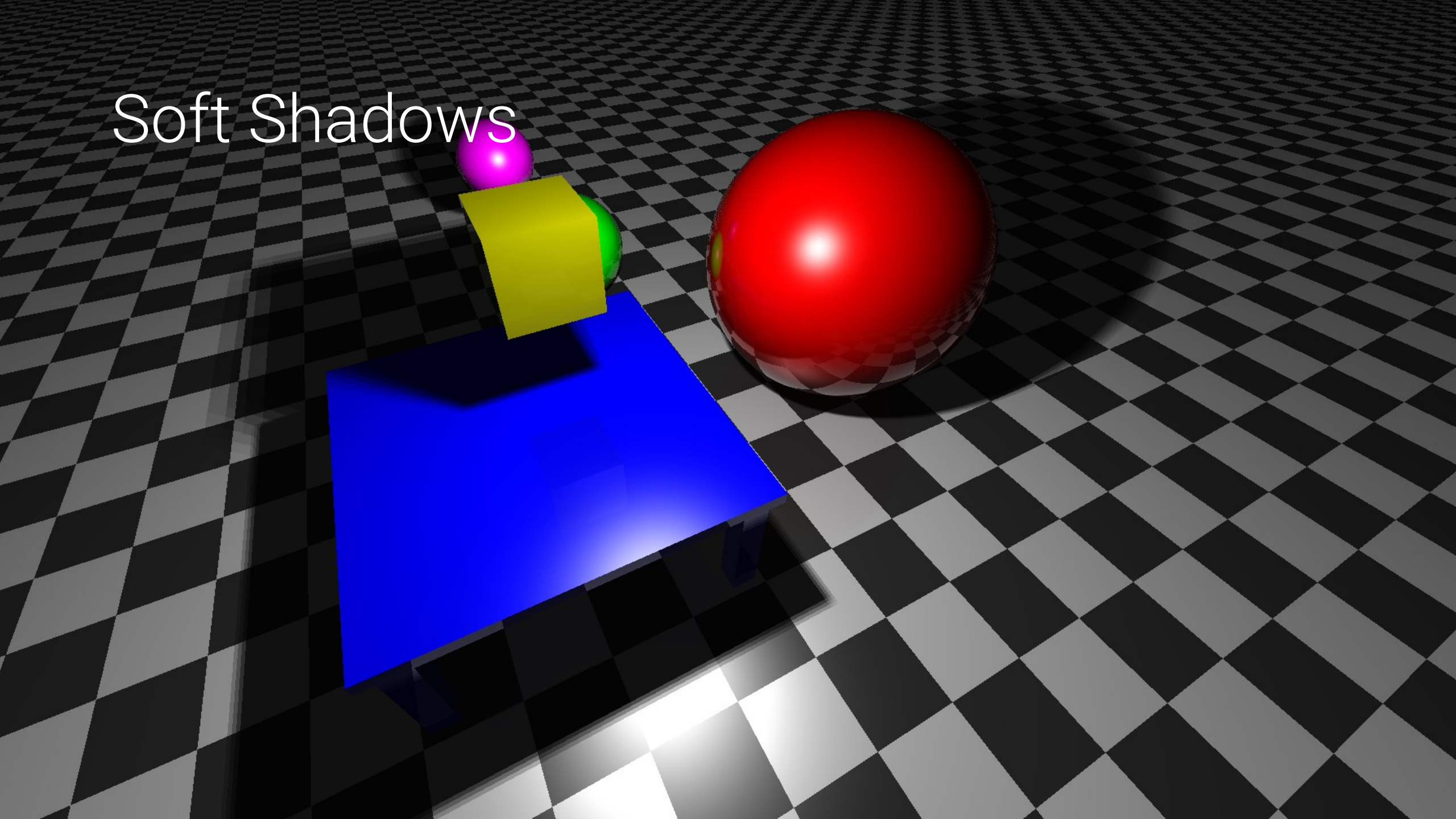


GLSL Code: Soft Shadows

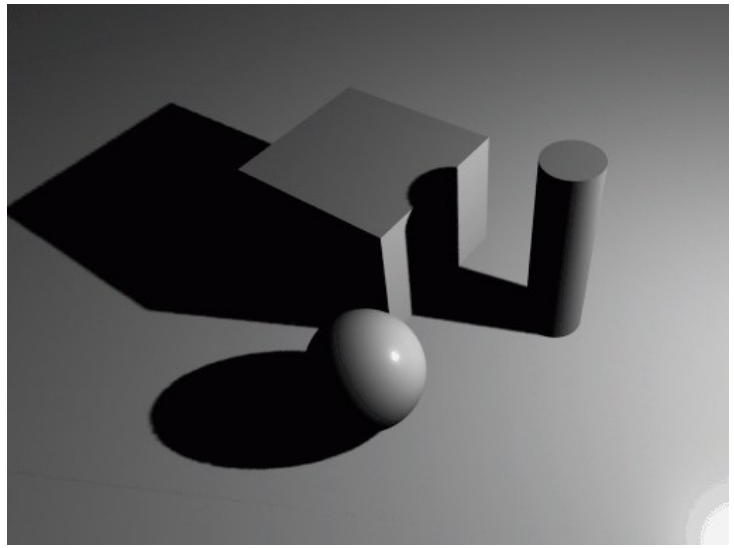
raytracing.fs.glsl

```
vec3 calcLightingSoftShadows(vec3 hitPoint, vec3 normal, vec3 inRay, vec3 color) {
    …
   const float delta = 2.0 / float(SHADOW_SAMPLES-1);
   for(float x=-1.0; x<=1.0; x+=delta){
     for(float y=-1.0; y<=1.0; y+=delta){
         vec3 nLightPos = lightPosition + vec3(x*LIGHT_SIZE,0.0,y*LIGHT_SIZE);
         vec3 lightVec = nLightPos - hitPoint;
         vec3 lightDir = normalize(lightVec);
         float lightDist = length(lightVec);
         vec3 hitNormal, hitColor;
         float shadowRayDist = rayTraceScene(hitPoint + lightDir*RAY_OFFSET, lightDir,
           hitNormal, hitColor );
         if(shadowRayDist < lightDist) {
           …
           shading += …
         }
         count += 1.0;
   } }
   return shading / count;
}
```

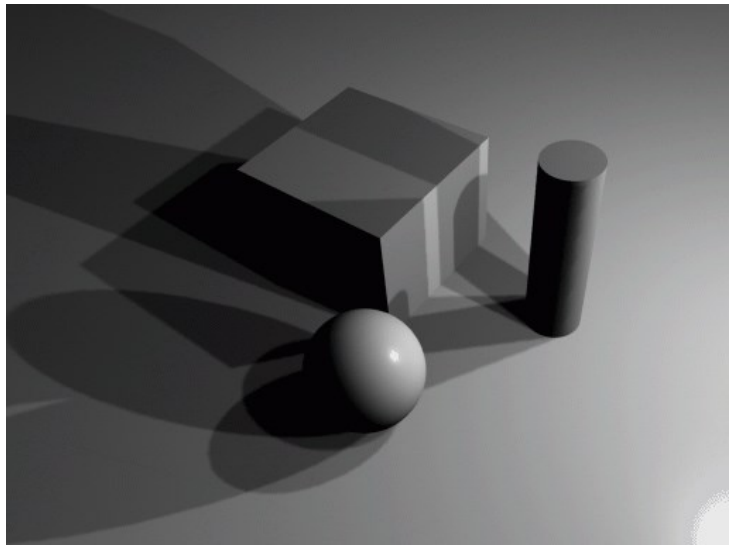
Soft Shadows



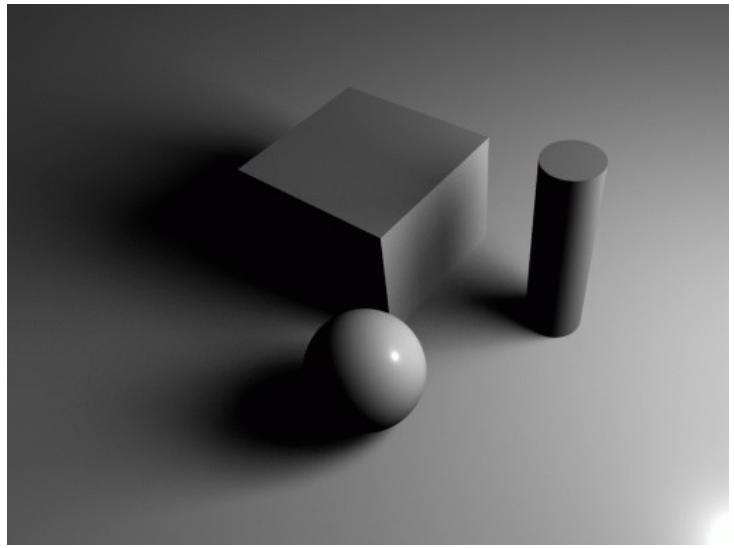
Soft Shadows: Sampling



1 sample



4 sample



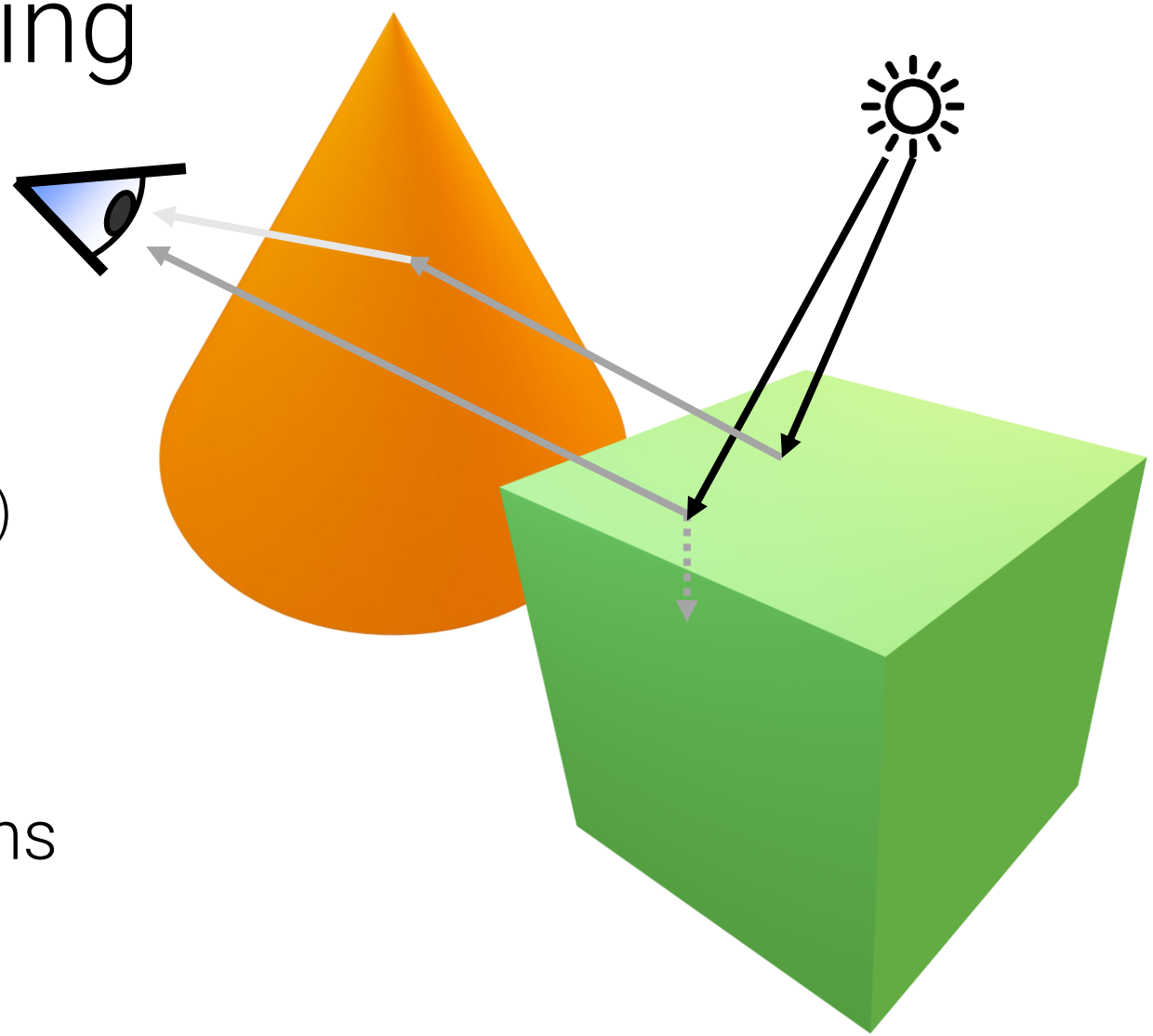
128 sample

Summary Raytracing

- Differs from Rasterization

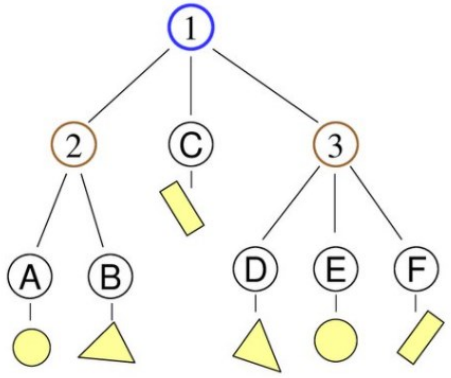
- Computes:
  - Detailed surfaces (if implicit)
  - Shadows and Soft Shadows
  - Correct reflections

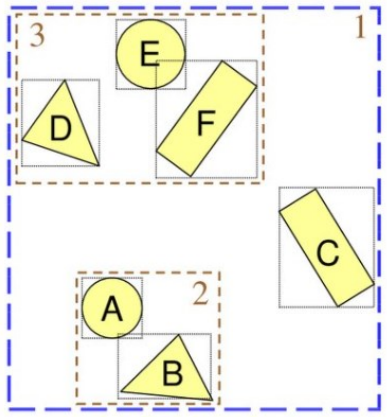
- Used for real-time reflections and shadows in games.

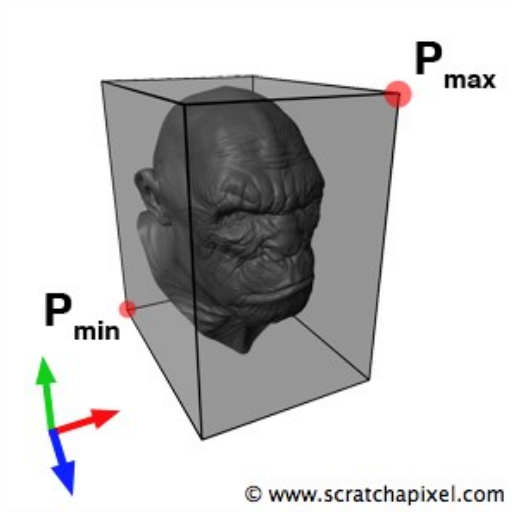


Outlook Raytracing

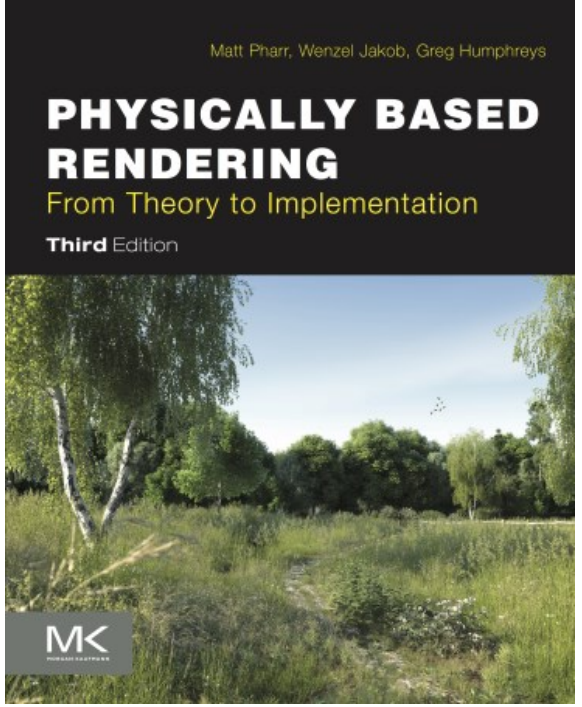
- Complex shading (PBR) possible, but expensive.
  Especially interaction of diffuse-to-diffuse surfaces, requires a lot of secondary rays (random Monte-Carlo sampling).
- Denoising, Filtering ...
- Optimization for multiple objects (triangles).
  Hierarchical grouping in trees:





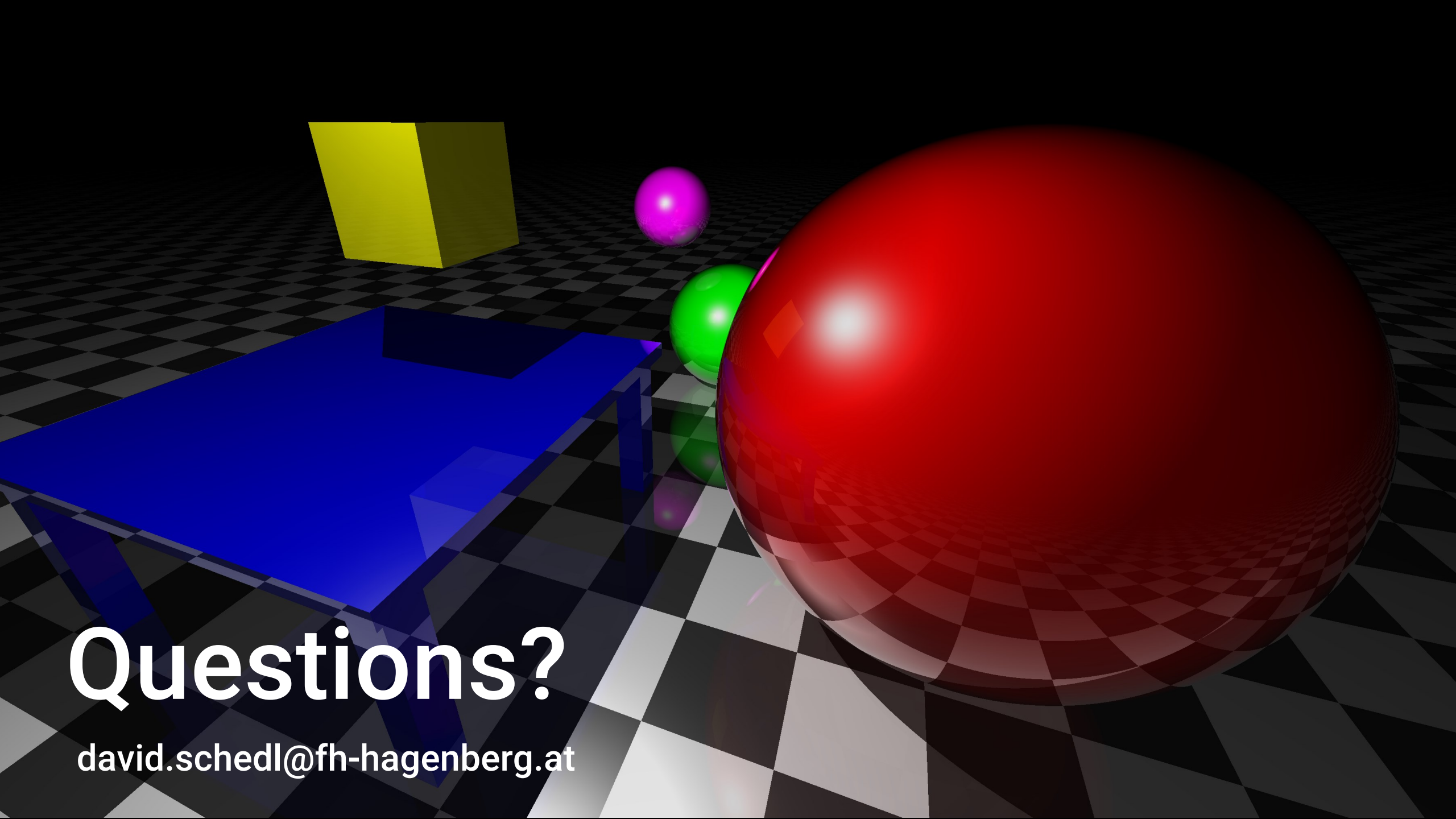


Useful Resources



- Physically Based Rendering Book (free version online): http://www.pbr-book.org/

- Chapter on Real-Time Raytracing (including basics): https://www.realtimerendering.com/Real-Time_Rendering_4th-Real-Time_Ray_Tracing.pdf


Questions?
david.schedl@fh-hagenberg.at

Exercise 06 –
Your Raytraced Scene

Create your own raytraced scene with new primitives and textures.

Define one surface pattern, like the checkerboard we used on the floor. For example, create lines, a color ramp, honeycombs, noise, or any other texture you can think of (but not a checkerboard or a solid color). If you want to explore this topic in more depth and want to get some inspiration search for keywords such as procedural textures or fractals.

Add at least one additional primitive that we did not cover in today's lab. For example, a cylinder, a torus, a cone, or any other structure that might be useful for your scene. You need to implement the intersection test and you need to compute normals for shading.
ADVANCED: you can implement a triangle primitive and use it to display a mesh. Note: loading a mesh can work but requires some effort. We simply define primitives in the rayTraceScene function, now.

Upload a PDF for the progress report.
Again, you can work in teams of 2 students.

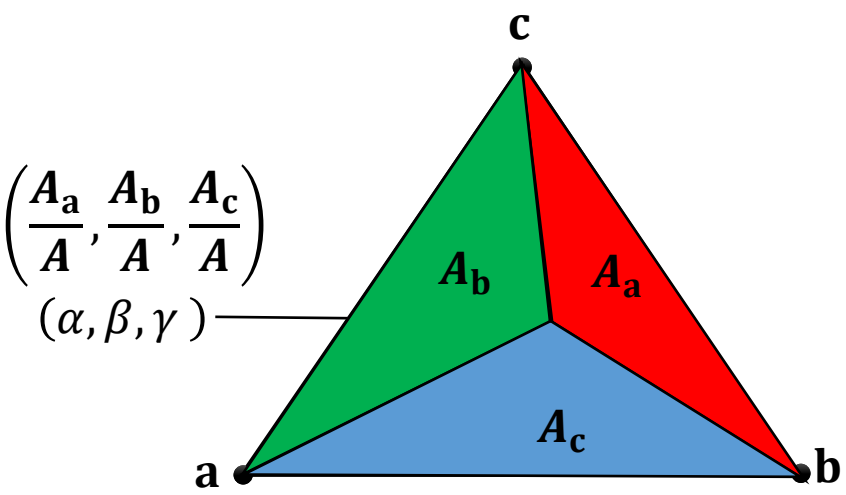
Some inspirations: https://www.iquilezles.org/www/articles/raymarchingdf/raymarchingdf.htm
(Note: raymarching not ray tracing)

Ray-triangle intersections

Ray-triangle intersection

- Given a triangle defined by the three points $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$
- Any point within the triangle can be defined by means of *barycentric coordinates* α, β, γ

  $\mathbf{p} = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$, with $\alpha + \beta + \gamma = 1$ and $\alpha, \beta, \gamma \geq 0$



Ray-triangle intersections

- *Möller et al.: Fast Minimum Storage Ray/Triangle Intersection*
- Barycentric:

$$\mathbf{p} = (1 - \beta - \gamma)\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$$

- Parametric ray:

$$\mathbf{p} = \mathbf{o} + t\mathbf{d}$$

$$(1 - \beta - \gamma)\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c} = \mathbf{o} + t\mathbf{d}$$

$$-t\mathbf{d} + \beta\mathbf{b} - \beta\mathbf{a} + \gamma\mathbf{c} - \gamma\mathbf{a} = \mathbf{o} - \mathbf{a}$$

$$[-\mathbf{d}, \mathbf{b} - \mathbf{a}, \mathbf{c} - \mathbf{a}][t, \beta, \gamma]^T = \mathbf{o} - \mathbf{a}$$

$$\begin{pmatrix} -d_x & b_x - a_x & c_x - a_x \\ -d_y & b_y - a_y & c_y - a_y \\ -d_z & b_z - a_z & c_z - a_z \end{pmatrix} \begin{pmatrix} t \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} o_x - a_x \\ o_y - a_y \\ o_z - a_z \end{pmatrix}$$

Ray-triangle intersections

- Solving the System of Equations

$$[-\mathbf{d}, \mathbf{b}-\mathbf{a}, \mathbf{c}-\mathbf{a}][t, \beta, \gamma]^T = \mathbf{o}-\mathbf{a}$$

$$\begin{bmatrix} t \\ \beta \\ \gamma \end{bmatrix} = \frac{1}{|-\mathbf{d}, \quad \mathbf{e}_1, \quad \mathbf{e}_2|} \begin{bmatrix} |\ \mathbf{s}, & \mathbf{e}_1, & \mathbf{e}_2| \\ |-\mathbf{d}, & \mathbf{s}, & \mathbf{e}_2| \\ |-\mathbf{d}, & \mathbf{e}_1, & \mathbf{s}\ | \end{bmatrix}$$

$$\mathbf{e}_1 = \mathbf{b}-\mathbf{a}, \mathbf{e}_2 = \mathbf{c}-\mathbf{a}, \mathbf{s} = \boldsymbol{o}-\boldsymbol{a}$$

$$\begin{bmatrix} t \\ \beta \\ \gamma \end{bmatrix} = \frac{1}{(\mathbf{d}\times\mathbf{e}_2)\cdot\mathbf{e}_1} \begin{bmatrix} (\mathbf{s}\times\mathbf{e}_1)\cdot\mathbf{e}_2 \\ (\mathbf{d}\times\mathbf{e}_2)\cdot\mathbf{s} \\ (\mathbf{s}\times\mathbf{e}_1)\cdot\mathbf{d} \end{bmatrix} = \frac{1}{\mathbf{g}\cdot\mathbf{e}_1} \begin{bmatrix} \mathbf{h}\cdot\mathbf{e}_2 \\ \mathbf{g}\cdot\mathbf{s} \\ \mathbf{h}\cdot\mathbf{d} \end{bmatrix}$$

$$where\ \mathbf{g} = (\mathbf{d}\times\mathbf{e}_2)\ and\ \mathbf{h} = (\mathbf{s}\times\mathbf{e}_1)$$

- Check $\beta, \gamma \in [0,1]$ and $\beta + \gamma \leq 1$ → valid intersection