Ray Tracing

Prof. Fabio Pellacini
[some original slides by Prof. Steve Marschner]

Approaches to rendering

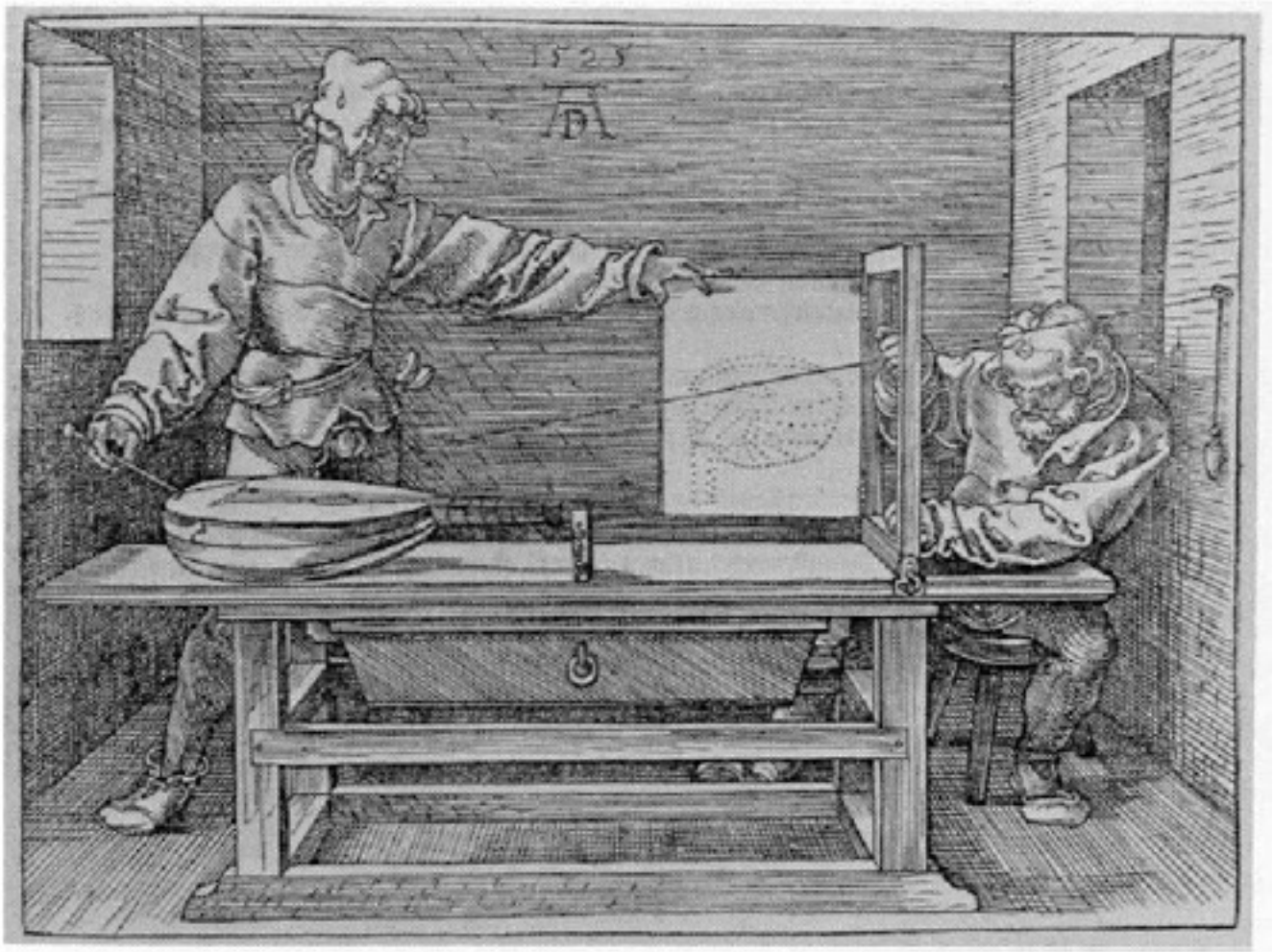
- **Rasterization**: project each object onto image
  - real-time rendering and GPUs

```
foreach object in scene:
  foreach pixel in image:
    if object affects pixel:
      do_something()
```

- **Raytracing**: project each pixel onto the objects
  - offline rendering, realistic rendering

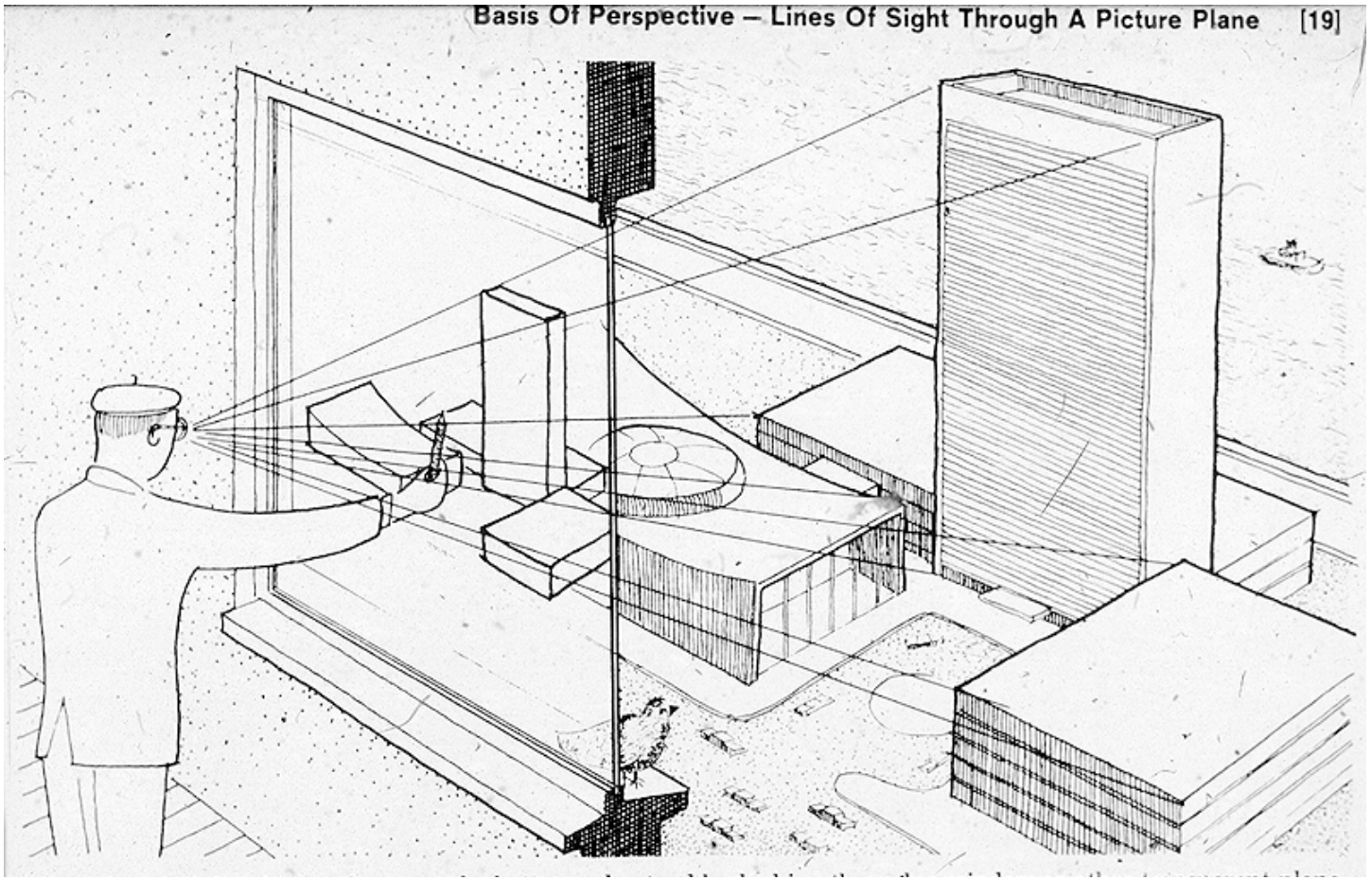
```
foreach pixel in image:
  foreach object in scene:
    if object affects pixel:
      do_something()
```


Idea Behind Raytracing



[Albrecht Dürer]

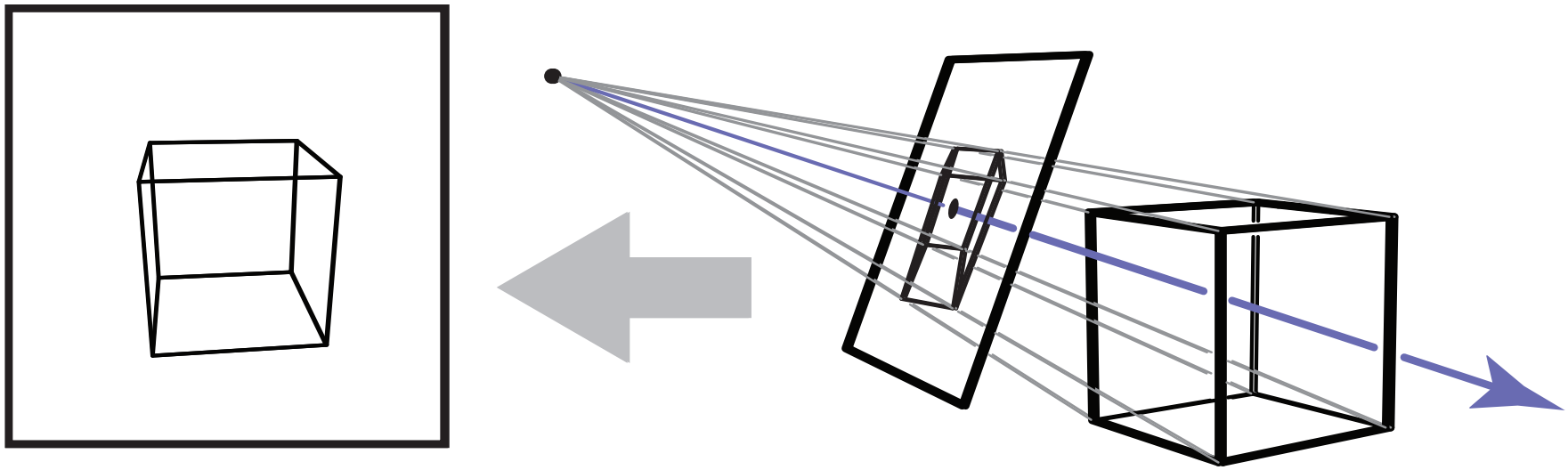
Idea Behind Raytracing



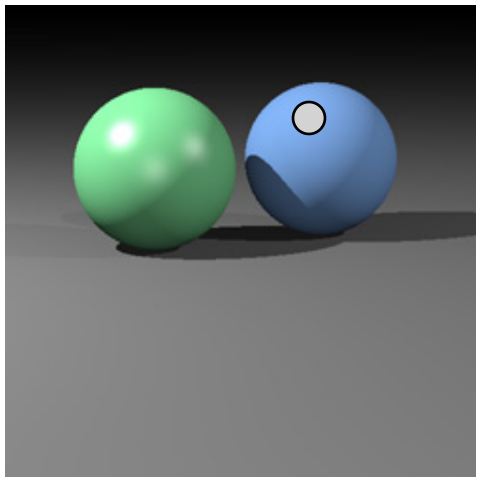
[source unknown]

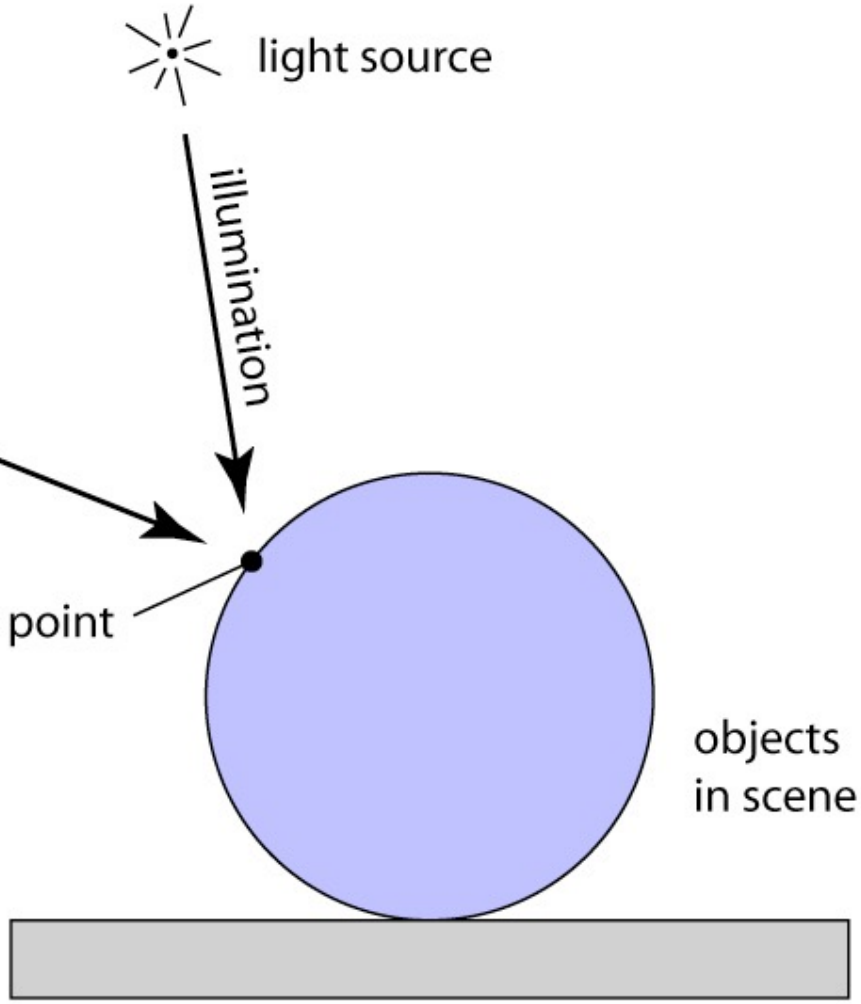
Ray tracing idea

- Start with a pixel, search for surface visible in that pixel
- Set of points that project to a point in the image: a ray



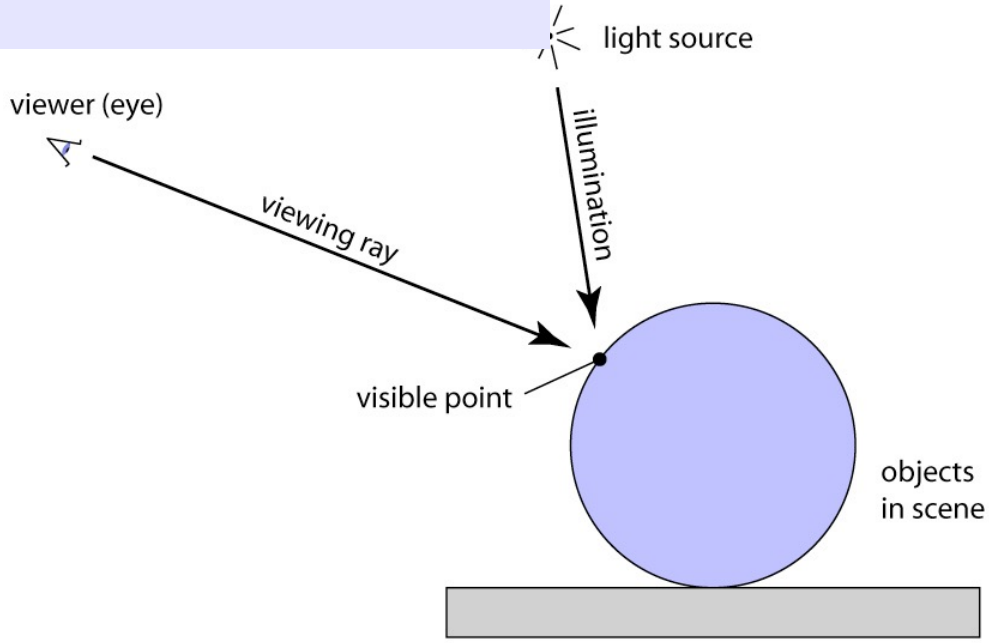
Ray tracing idea





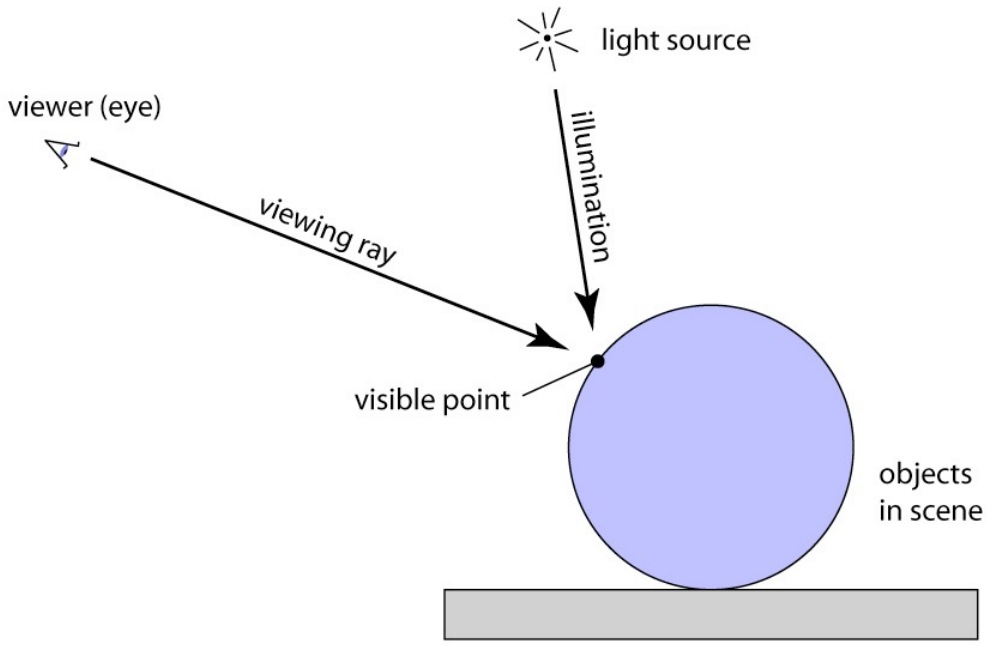
Ray tracing algorithm

```
for(auto j : range(img.height)) {
  for(auto i : range(img.width)) {
    auto uv = (vec2f{i,j} + 0.5f) / img.size;
    auto ray = camera_ray(cam, uv);
    img[i,j] = shade(scene, ray);
  }
}
```



Ray tracing algorithm

```
vec3f shade(scene* scene, ray3f ray) {
  auto point = intersect(scene, ray)
  Return compute_illumination(point)
}
```



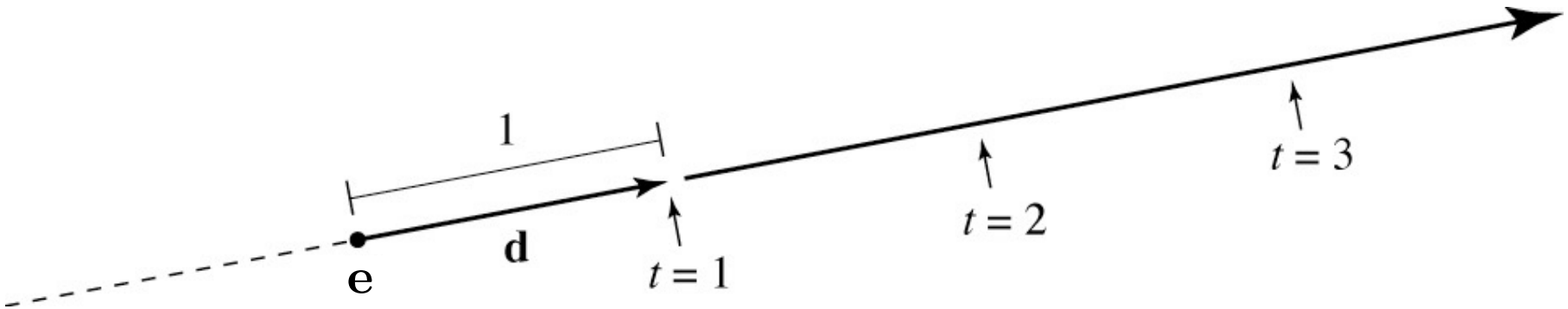
Camera Rays

Ray: a half line

- Standard representation: point **e** and direction **d**
  - this is a parametric equation for the line
  - add min and max values for *t*

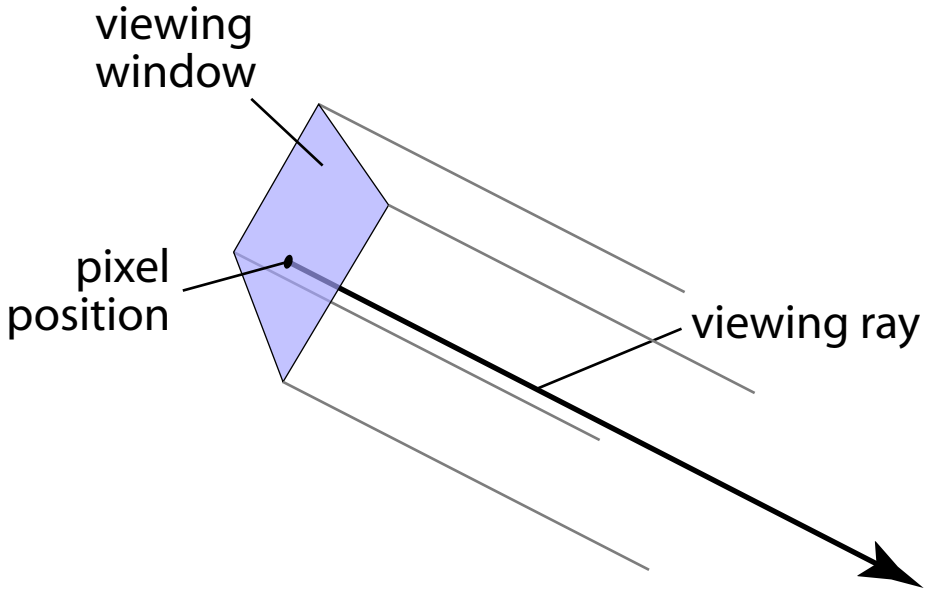
```
struct ray3f {
  vec3f e, d;
  float tmin, tmax;
}
```

$$\mathbf{p}(t) = \mathbf{e} + t\mathbf{d}$$



Orthographic projection

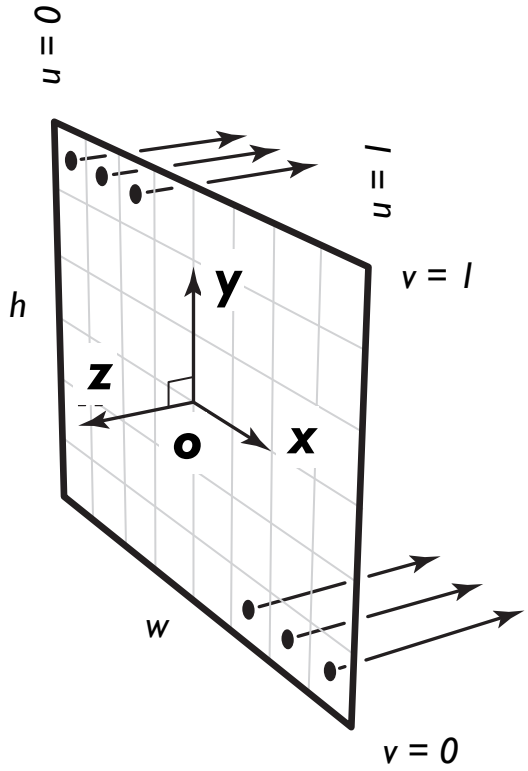
- Ray origin (varying): pixel position on viewing window
- Ray direction (constant): view direction



Orthographic projection – rays

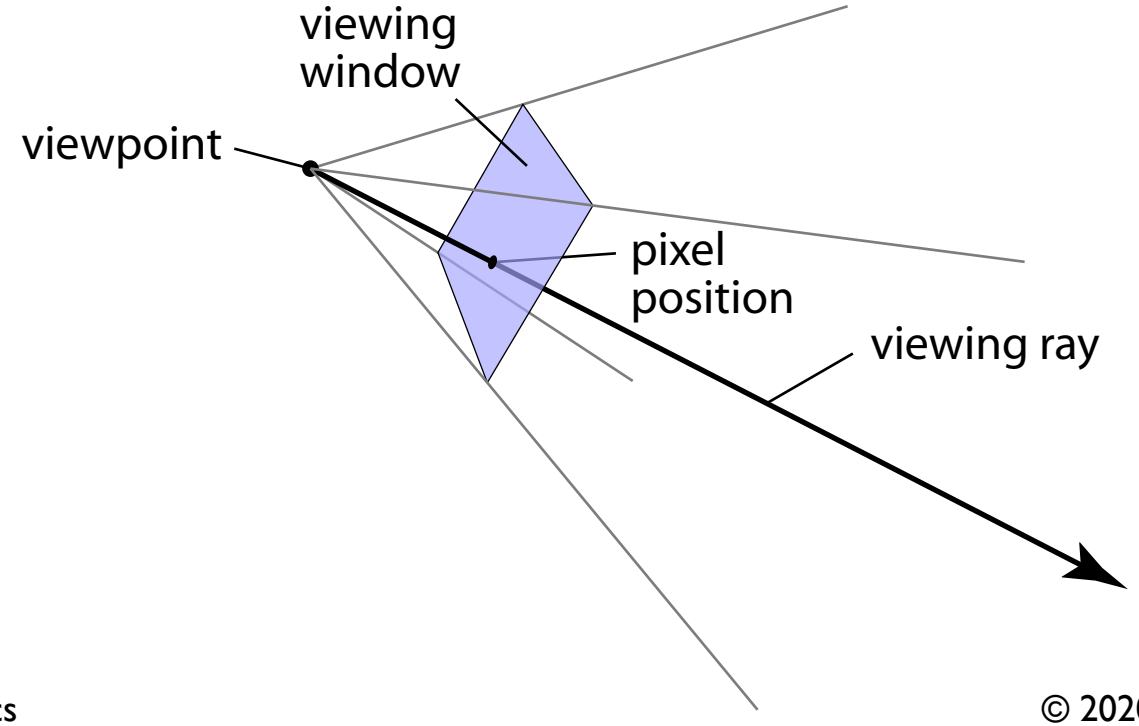
- Position and orient the camera with a frame $F = \{\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{o}\}$
- Image plane of size *(w, h)* parametrized by *(u, v)* in [0,1]^2
- Define point on image plane **q** and with that the ray **r** as

$$\mathbf{q} = \mathbf{o} + (u - 0.5)w\mathbf{x} + (v - 0.5)h\mathbf{y}$$
$$\text{ray}(u, v) = \{\mathbf{q}, -\mathbf{z}\}$$
$$\text{with } (u, v) \in [0, 1]^2$$



Perspective projection – rays

- Use window analogy directly
- Ray origin (constant): viewpoint
- Ray direction (varying): toward pixel position on viewing window



Perspective projection – rays

- Position and orient the camera with a frame $F = \{\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{o}\}$
- Image of size *(w, h) at* distance *d* parametrized by *(u, v)* in [0,1]^2
- Use field of view *fov* and aspect ratio *r* to define image size

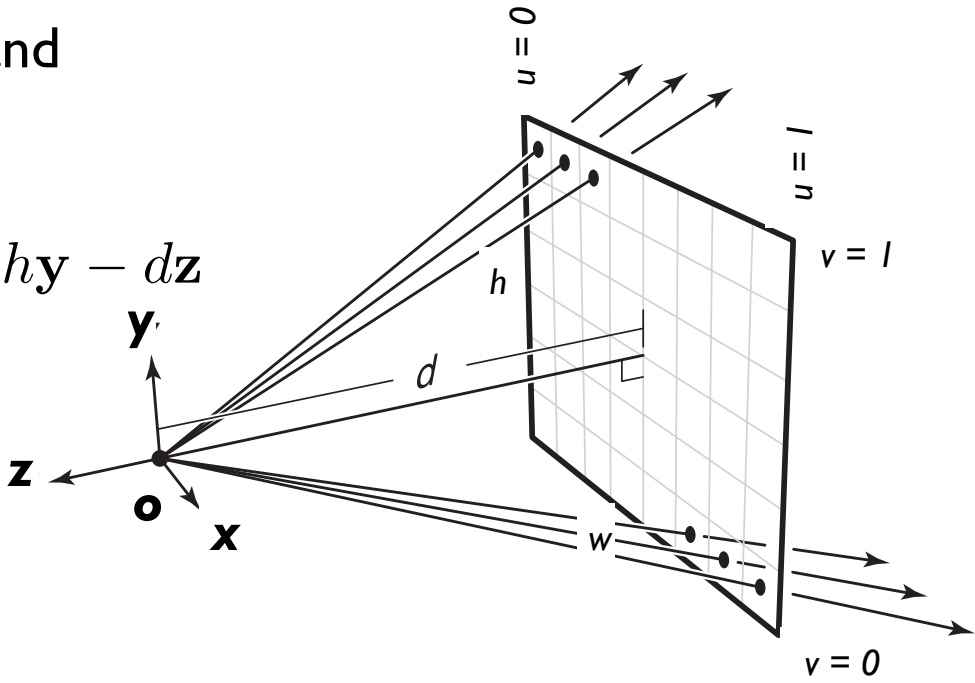
$$w = hr \quad h = 2d\tan(fov/2)$$

- Define point on image plane **q** and with that the ray **r** as

$$\mathbf{q} = \mathbf{o} + (u - 0.5)w\mathbf{x} + (v - 0.5)h\mathbf{y} - d\mathbf{z}$$

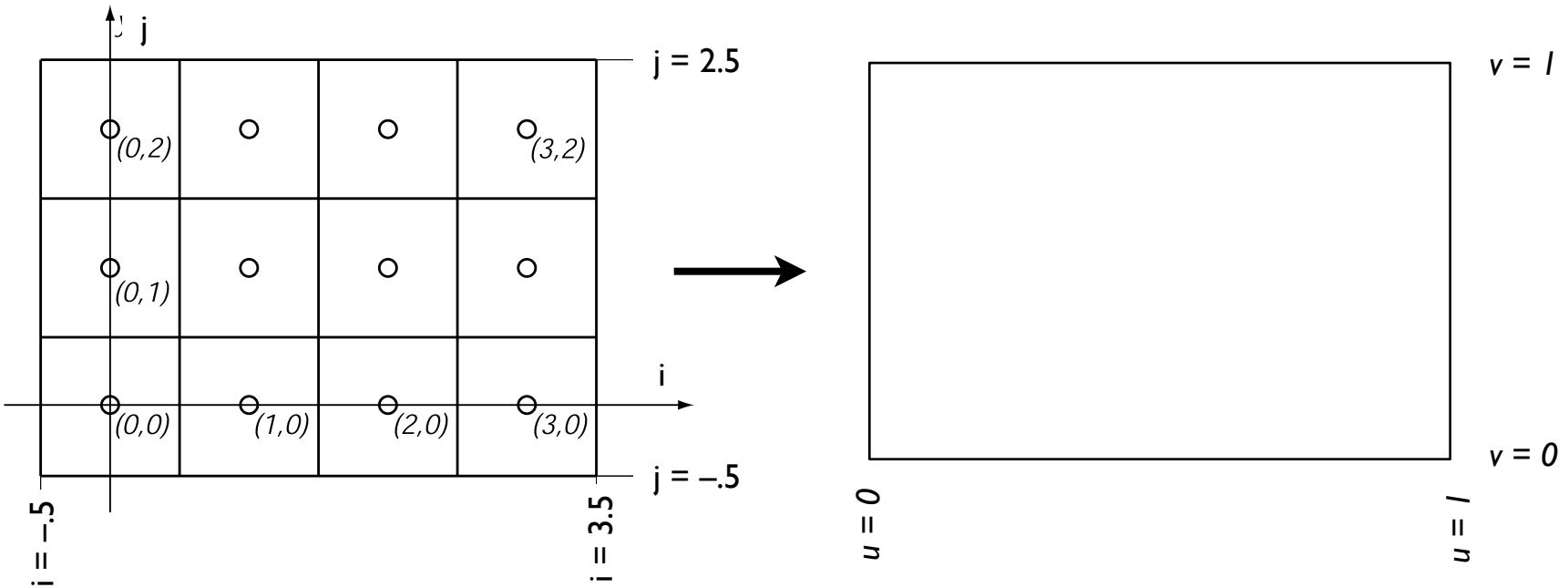
$$\text{ray}(u, v) = \left\{\mathbf{o}, \frac{\mathbf{q} - \mathbf{o}}{|\mathbf{q} - \mathbf{o}|}\right\}$$

$$\text{with } (u, v) \in [0, 1]^2$$



Pixel-to-image mapping

- Map pixels *(i,j)* to image plane coordinates *(u,v)*

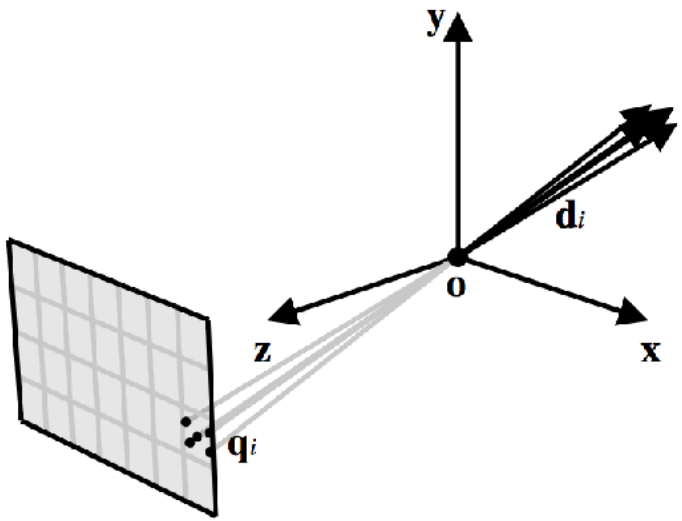


$$(u, v) = \left(\frac{i + 0.5}{n_x}, \frac{j + 0.5}{n_y} \right) \text{ with } (i, j) \in [0, n_x - 1] \times [0, n_y - 1]$$

Pinhole camera

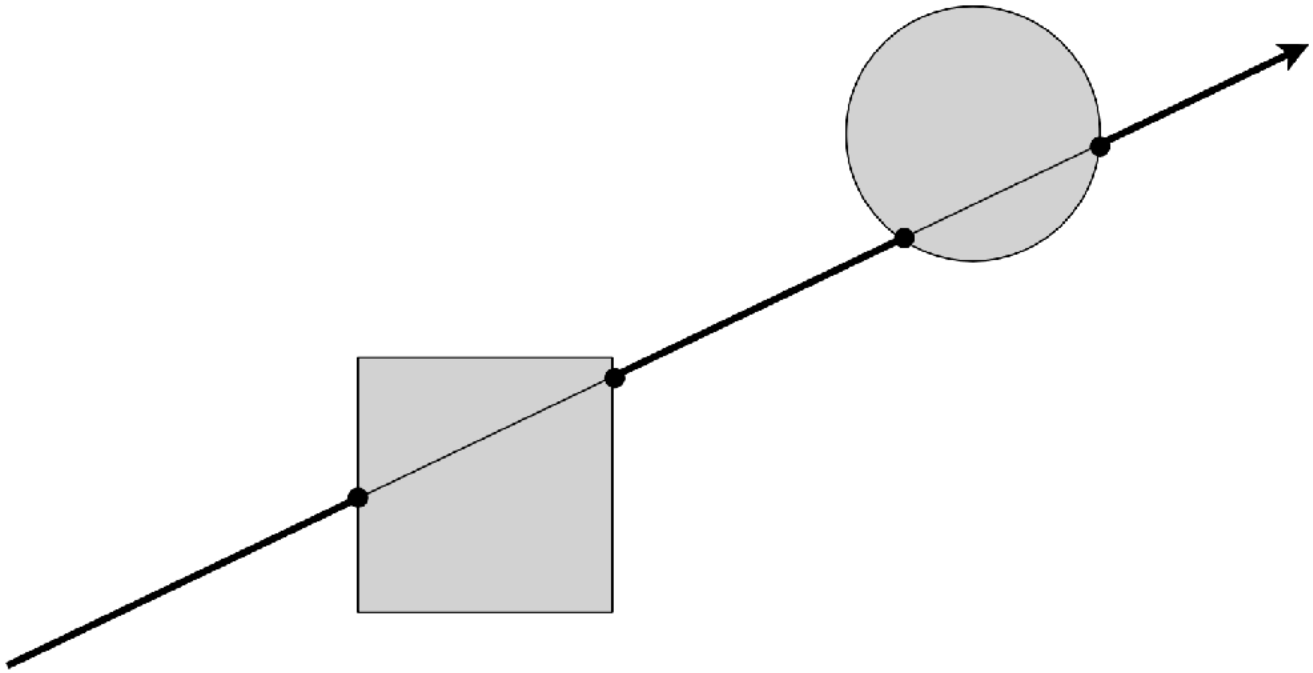
- Model an idealized pinhole camera with the sensor at the back. Same as the perspective model, but closer to real cameras.

```
ray3f eval_camera(camera* camera, vec2f image_uv) {
  auto q = vec3f{camera->film.x * (0.5 - image_uv.x),
    camera->film.y * (image_uv.y - 0.5), camera->lens};
  auto e = vec3f{0}; auto d = normalize(-q - e);
  return ray3f{transform_point(camera->frame, e),
      transform_direction(camera->frame, d)};
}
```



Ray-Scene Intersection

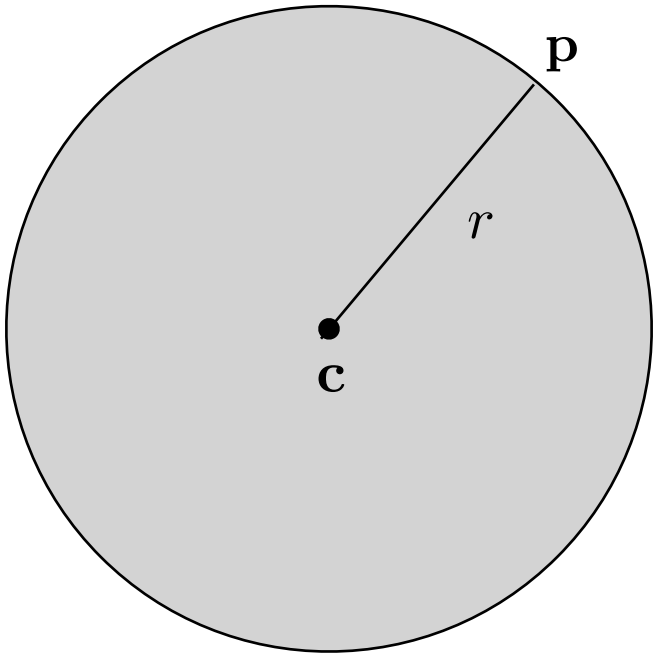
Ray intersection



Ray-sphere intersection

- Points on a sphere at at equal distance from its center

$$|\mathbf{p} - \mathbf{c}|^2 = r^2$$



Ray-sphere intersection

- Enforce point on a ray and on a sphere
- Leads to system of equations

$$\begin{cases} \mathbf{p}(t) = \mathbf{e} + t\mathbf{d} \\ |\mathbf{p} - \mathbf{c}|^2 = r^2 \end{cases}$$

- By substitution

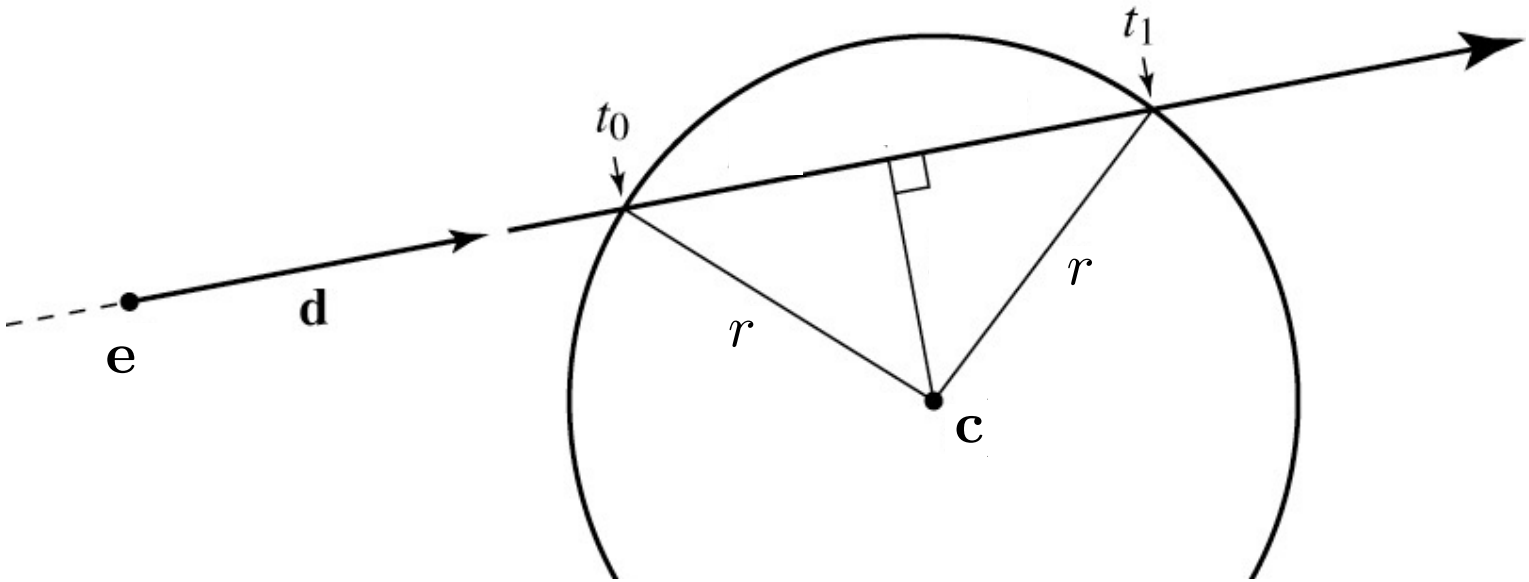
$$(\mathbf{e} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{e} + t\mathbf{d} - \mathbf{c}) = r^2$$

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

Ray-sphere intersection

- Algebraic equation $at^2 + bt + c = 0$
- With $a = |\mathbf{d}|^2 \quad b = 2\mathbf{d} \cdot (\mathbf{e} - \mathbf{c}) \quad c = |\mathbf{e} - \mathbf{c}|^2 - r^2$
- No solution for $d = b^2 - 4ac < 0$
- Two solutions

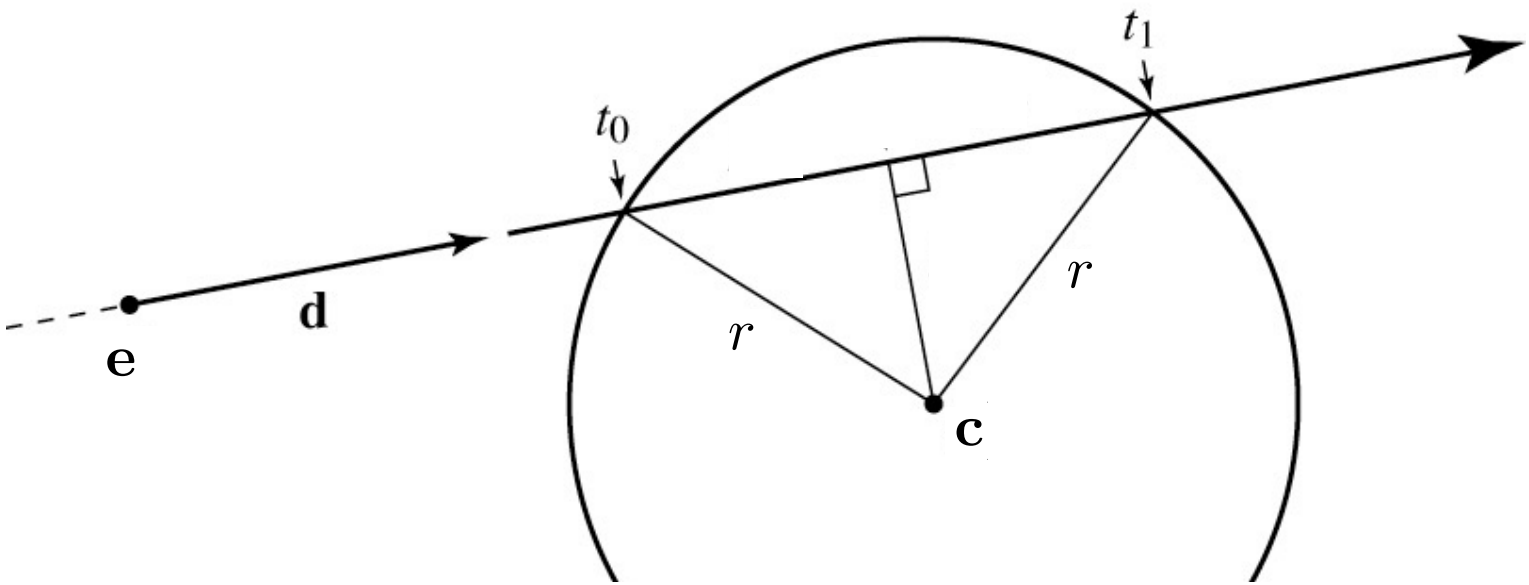
$$t_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$



Ray-sphere intersection

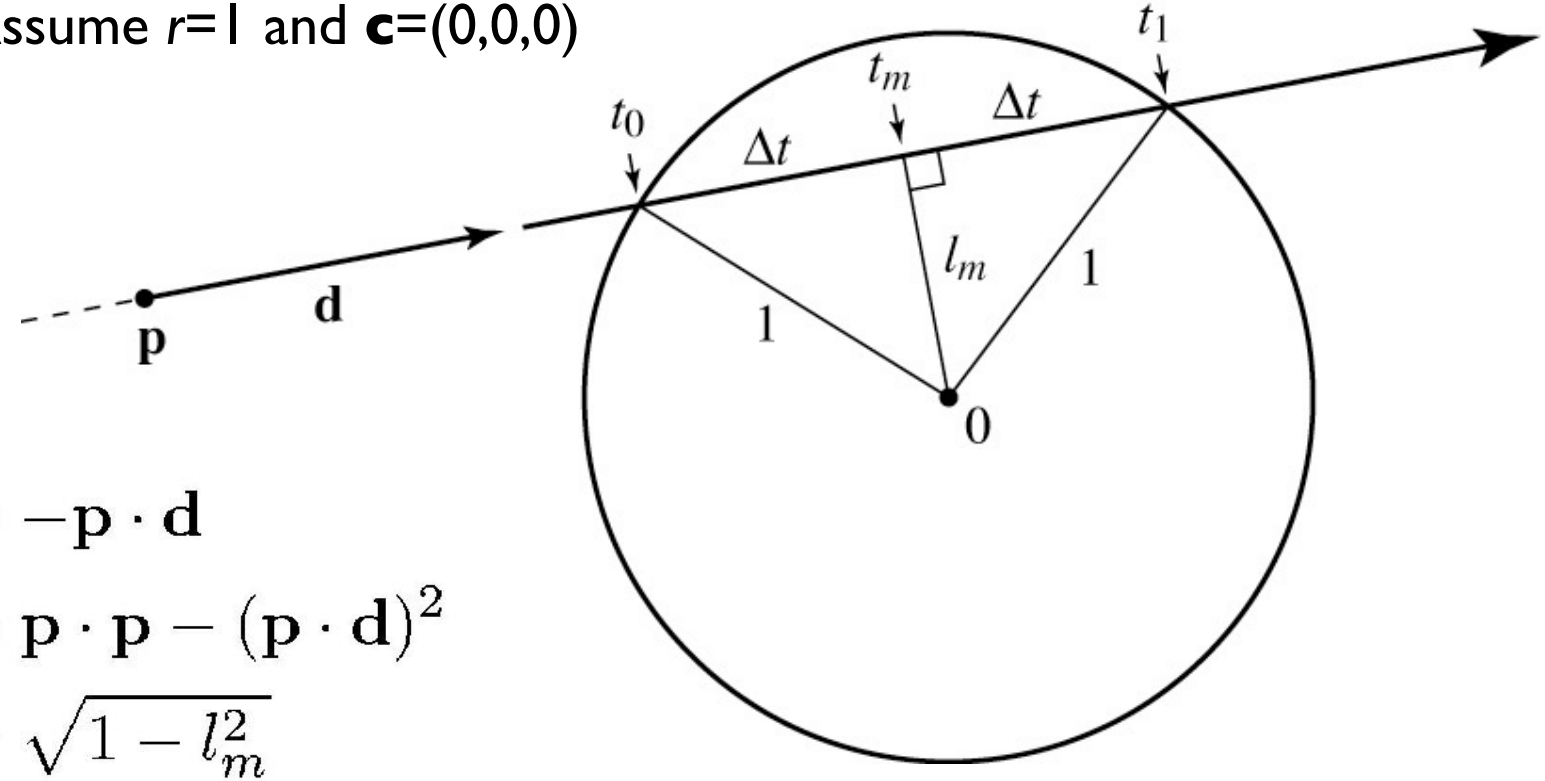
- Pick closest intersection within the ray bounds

$$t_{min} \leq t_{\pm} \leq l_m ax$$



Ray-sphere geometric

- Assume *r*=1 and **c**=(0,0,0)



$$t_m = -\mathbf{p} \cdot \mathbf{d}$$

$$l_m^2 = \mathbf{p} \cdot \mathbf{p} - (\mathbf{p} \cdot \mathbf{d})^2$$

$$\Delta t = \sqrt{1 - l_m^2}$$

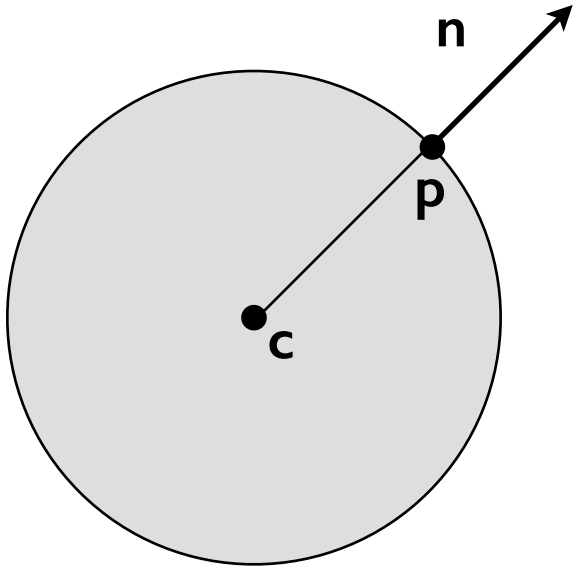
$$= \sqrt{(\mathbf{p} \cdot \mathbf{d})^2 - \mathbf{p} \cdot \mathbf{p} + 1}$$

$$t_{0,1} = t_m \pm \Delta t = -\mathbf{p} \cdot \mathbf{d} \pm \sqrt{(\mathbf{p} \cdot \mathbf{d})^2 - \mathbf{p} \cdot \mathbf{p} + 1}$$

Sphere normals

- Shading depends on geometric normals
- For spheres, the normal is along the line that connects the sphere point to the sphere center

$$\mathbf{n} = \frac{\mathbf{p} - \mathbf{c}}{|\mathbf{p} - \mathbf{c}|}$$



Ray-triangle intersection

- Use algebraic view since it is significantly simpler
- Write point w.r.t. barycentric coordinates and save for those
- Also used later for texturing

Ray-triangle intersection

- Every point on the triangle can be as

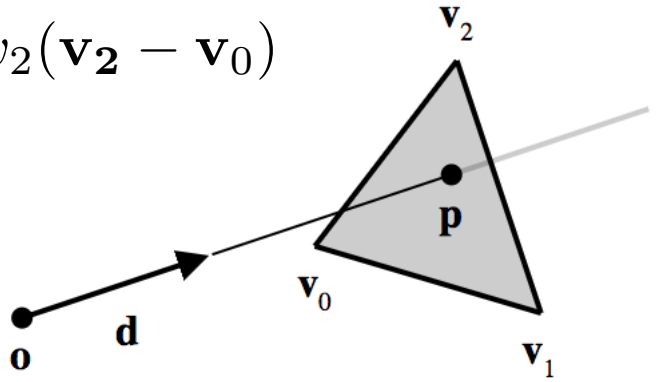
$$\mathbf{p}(w_1, w_2) = \mathbf{v}_0 + w_1(\mathbf{v}_1 - \mathbf{v}_0) + w_2(\mathbf{v}_2 - \mathbf{v}_0)$$

- Set up a system of equations

$$\begin{cases} \mathbf{p}(t) = \mathbf{e} + \mathbf{d}t \\ \mathbf{p}(w_1, w_2) = \mathbf{v}_0 + w_1(\mathbf{v}_1 - \mathbf{v}_0) + w_2(\mathbf{v_2} - \mathbf{v}_0) \end{cases}$$

- By substitution

$$\mathbf{e} + \mathbf{d}t = \mathbf{v}_0 + w_1(\mathbf{v}_1 - \mathbf{v}_0) + w_2(\mathbf{v_2} - \mathbf{v}_0)$$



Ray-triangle intersection

- Linear system of 3 equations for 3 unknowns

$$\mathbf{e} + \mathbf{d}t = \mathbf{v}_0 + w_1(\mathbf{v}_1 - \mathbf{v}_0) + w_2(\mathbf{v}_2 - \mathbf{v}_0)$$

$$w_1(\mathbf{v}_0 - \mathbf{v}_1) + w_2(\mathbf{v}_0 - \mathbf{v}_2) + \mathbf{d}t = \mathbf{v}_0 - \mathbf{e}$$

$$\begin{bmatrix} \mathbf{v}_0 - \mathbf{v}_1 & \mathbf{v}_0 - \mathbf{v}_2 & \mathbf{d} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ t \end{bmatrix} = \begin{bmatrix} \mathbf{v}_0 - \mathbf{e} \end{bmatrix}$$

- Solve with Cramer's rule

$$t = \frac{(\mathbf{v}_2 \times \mathbf{e}_1) \cdot \mathbf{e}_2}{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{e}_1} \quad w_1 = \frac{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{v}_2}{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{e}_1} \quad w_2 = \frac{(\mathbf{v}_2 \times \mathbf{e}_1) \cdot \mathbf{d}}{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{e}_1}$$

$$\text{with} \quad \mathbf{v}_2 = \mathbf{e} - \mathbf{v}_0 \quad \mathbf{e}_1 = \mathbf{v}_1 - \mathbf{v}_0 \quad \mathbf{e}_2 = \mathbf{v}_2 - \mathbf{v}_0$$

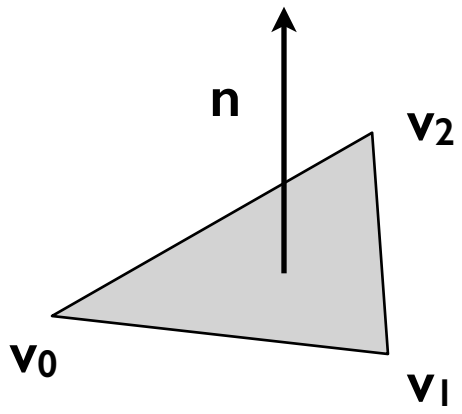
- Check for constraints on barycentric coordinates and ray distance

$$t_{min} \leq t \leq t_{max} \quad 0 \leq w_1 \leq 1 \quad 0 \leq w_2 \leq 1 \quad w_1 + w_2 \leq 1$$

Triangle normal

- Shading depends on geometric normals
- For triangles, the normal is the normal of the plane where the triangle lies
- Normals can computed from the triangle vertices as

$$\mathbf{n} = \frac{(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)}{|(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)|}$$

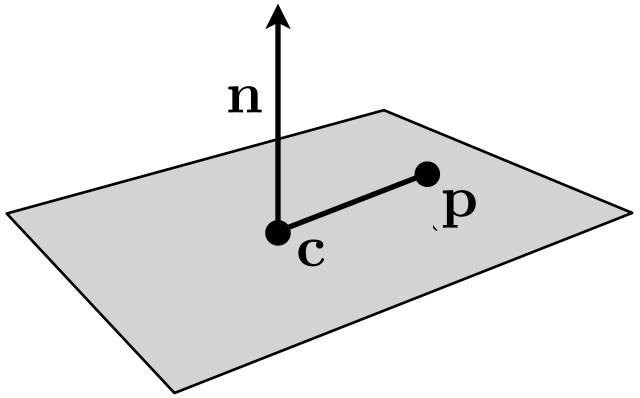


Infinite plane representation

- Vector between two points on the plane is orthogonal to the plane normal

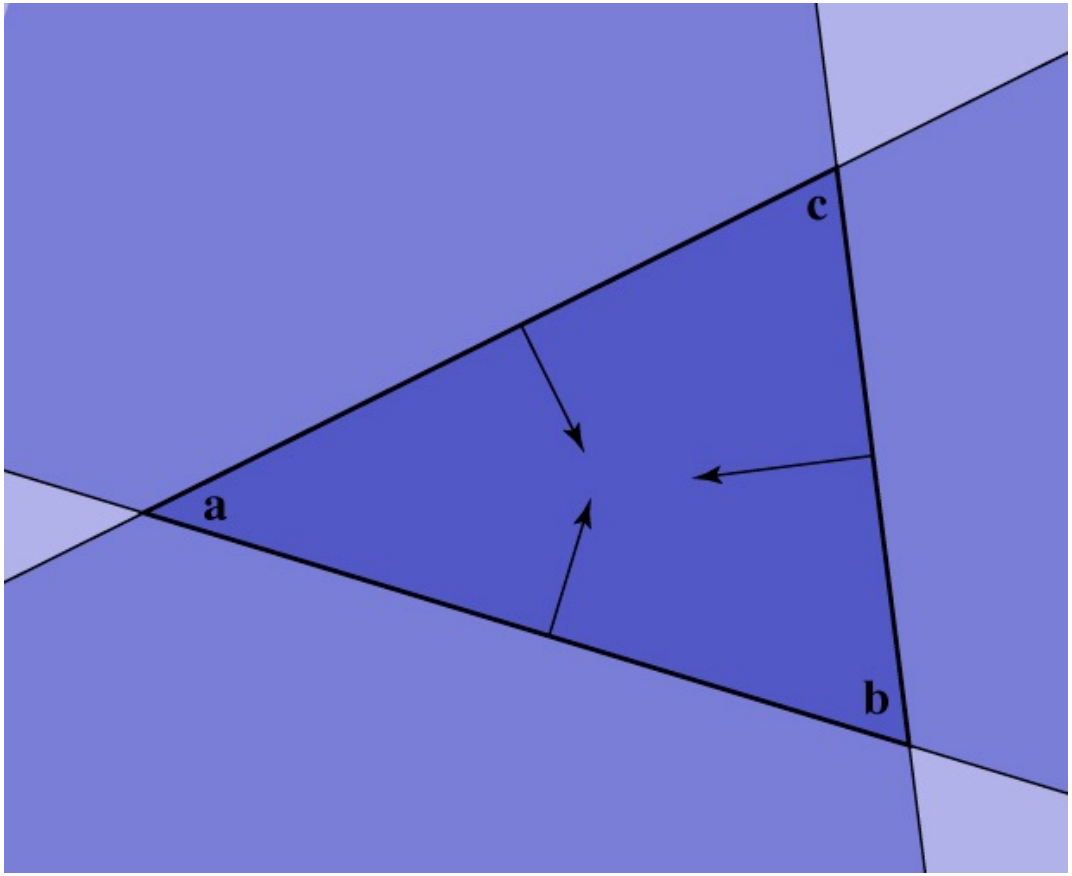
$$(\mathbf{p} - \mathbf{c}) \cdot \mathbf{n} = 0$$

$$\mathbf{p} \cdot \mathbf{n} + k = 0$$



Ray-triangle geometric

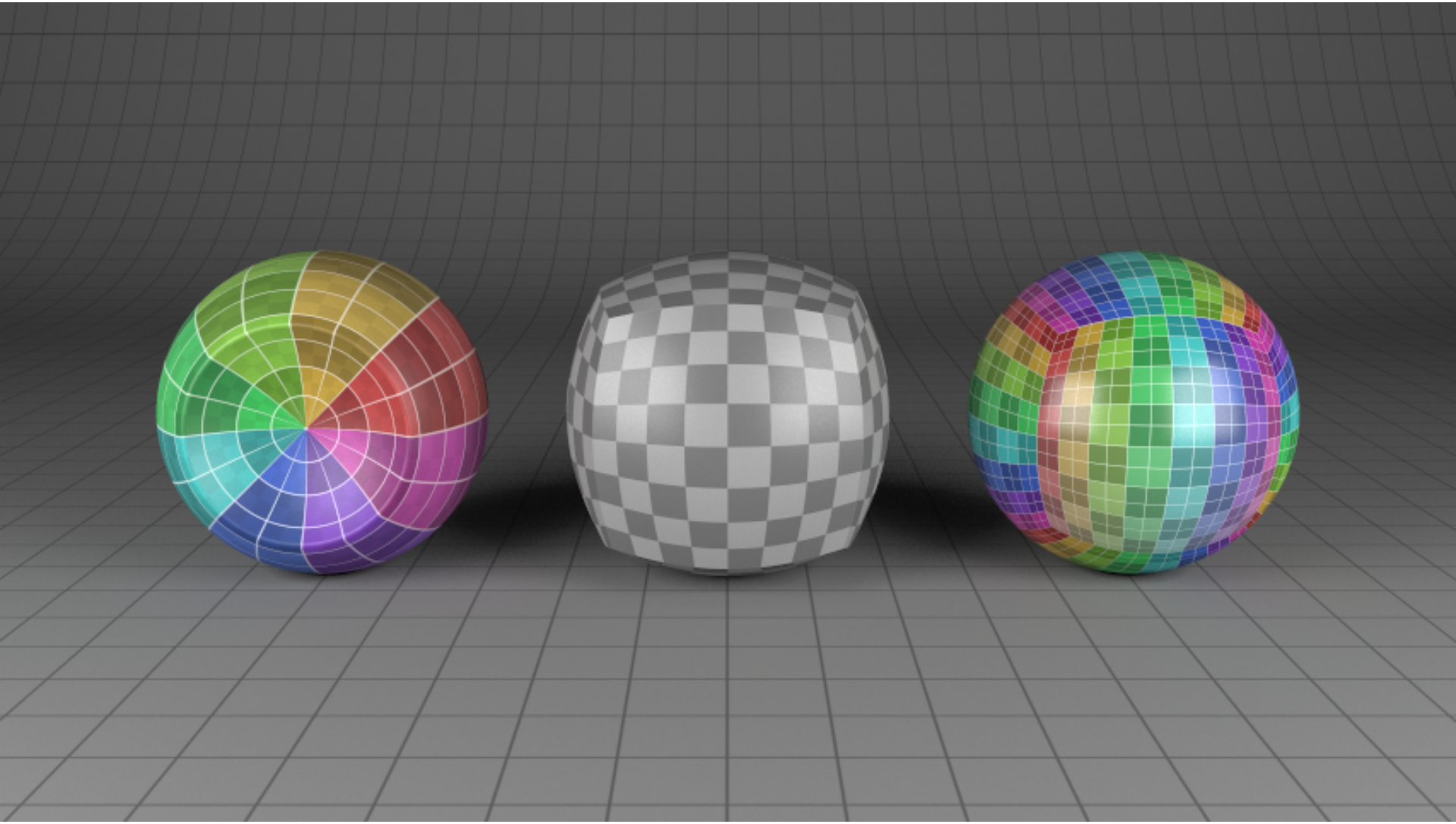
- First compute the intersection with the infinite plane
- In plane, triangle is the intersection of 3 half spaces



Ray-triangle intersection

```
bool intersect_triangle(ray3f ray, vec3f p0, vec3f p1,
  vec3f p2, vec2f* uv, float* dist) {
  // compute triangle edges
  auto edge1 = p1 - p0; auto edge2 = p2 - p0;
  // compute determinant to solve a linear system
  auto pvec = cross(ray.d, edge2); auto det = dot(edge1, pvec);
  // check determinant and exit if triangle and ray are parallel
  if (det == 0) return false; // can use epsilon
  auto idet = 1.0f / det;
  // compute and check first barycentric coordinate
  auto tvec = ray.o - p0;  auto u = dot(tvec, pvec) * idet;
  if (u < 0 || u > 1) return false;
  // compute and check second barycentric coordinate
  auto qvec=cross(tvec,edge1); auto v = dot(ray.d, qvec) * idet;
  if (v < 0 || u + v > 1) return false;
  // compute and check ray parameter
  auto t = dot(edge2, qvec) * inv_det;
  if (t < ray.tmin || t > ray.tmax) return false;
  // intersection occurred: set params and exit
  *uv = {u, v}; *dist = t; return true;
}
```

Ray-triangle intersection

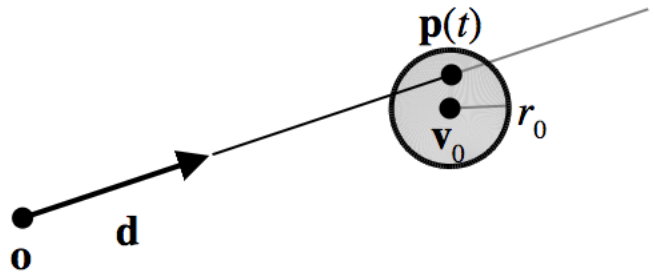


Ray-point intersection

- *Approximate* solution since points are really unidimensional
- Find ray-point distance and then check whether it is less then segment radius

$$\frac{d}{dt}D^2(t) = \frac{d}{dt}|(\mathbf{e}+t\mathbf{d})-\mathbf{v}_0|^2 = 0 \rightarrow$$
$$2\mathbf{d}\cdot(\mathbf{e}+t\mathbf{d}-\mathbf{v}_0) = 0 \rightarrow$$
$$t = \frac{(\mathbf{v}_0-\mathbf{o})\cdot\mathbf{d}}{\mathbf{d}\cdot\mathbf{d}}$$

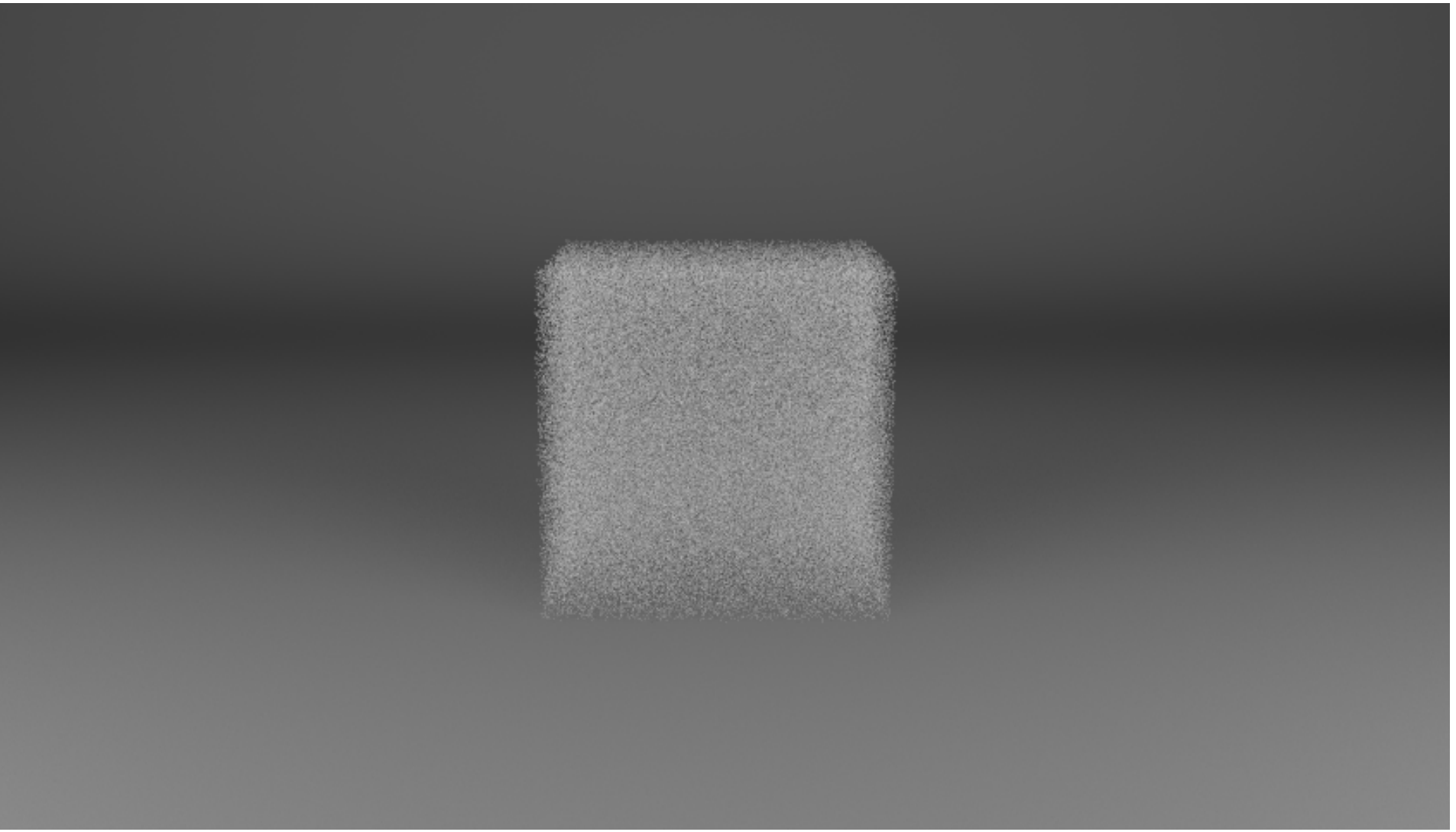
- Then clamp *t* in the valid range and check with point radius



Ray-point intersection

```
bool intersect_point(ray3f ray, vec3f p, float r,
  vec2f* uv, float* dist) {
  // find parameter for line-point minimum distance
  auto w = p - ray.o;
  auto t = dot(w, ray.d) / dot(ray.d, ray.d);
  // exit if not within bounds
  if (t < ray.tmin || t > ray.tmax) return false;
  // test for line-point distance vs point radius
  auto rp  = ray.o + ray.d * t;
  auto prp = p - rp;
  if (dot(prp, prp) > r * r) return false;
  // intersection occurred: set params and exit
  *uv = {0, 0}; *dist = t; return true;
}
```

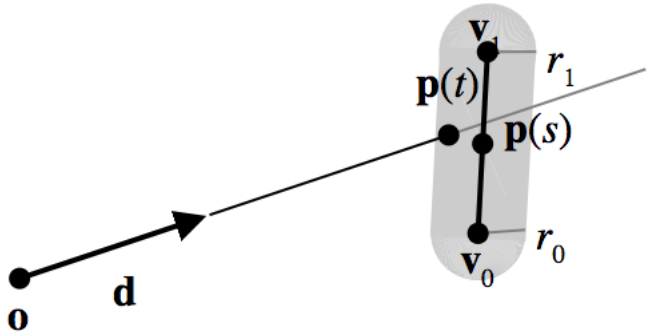

Ray-point intersection



Ray-segment intersection

- *Approximate* solution since lines are really unidimensional
- Find the closest points on ray and segment and then check whether their distance is less then segment radius
- Compute the points by minimizing the distance over the ray and segment parameters

$$D^2(s,t) = |(\mathbf{e} + t\mathbf{d}) - (\mathbf{v}_0 + s(\mathbf{v}_1 - \mathbf{v}_0))|^2 \rightarrow (s,t) = \arg\min D^2(s,t)$$



Ray-line intersection

- Solve by setting derivatives to zero

$$\begin{cases} \partial D^2(s,t)/\partial s = 0 \\ \partial D^2(s,t)/\partial t = 0 \end{cases} \rightarrow$$

$$\begin{cases} 2(\mathbf{v}_1 - \mathbf{v}_0) \cdot [(\mathbf{e} + t\mathbf{d}) - (\mathbf{v}_0 + s(\mathbf{v}_1 - \mathbf{v}_0))] = 0 \\ 2\mathbf{d} \cdot [(\mathbf{e} + t\mathbf{d}) - (\mathbf{v}_0 + s(\mathbf{v}_1 - \mathbf{v}_0))] = 0 \end{cases} \rightarrow$$

$$s = \frac{be - cd}{ac - b^2} \quad t = \frac{ae - bd}{ac - b^2} \quad \text{with}$$

$$a = \mathbf{d} \cdot \mathbf{d} \quad b = \mathbf{d} \cdot (\mathbf{v}_1 - \mathbf{v}_0) \quad c = (\mathbf{v}_1 - \mathbf{v}_0) \cdot (\mathbf{v}_1 - \mathbf{v}_0)$$

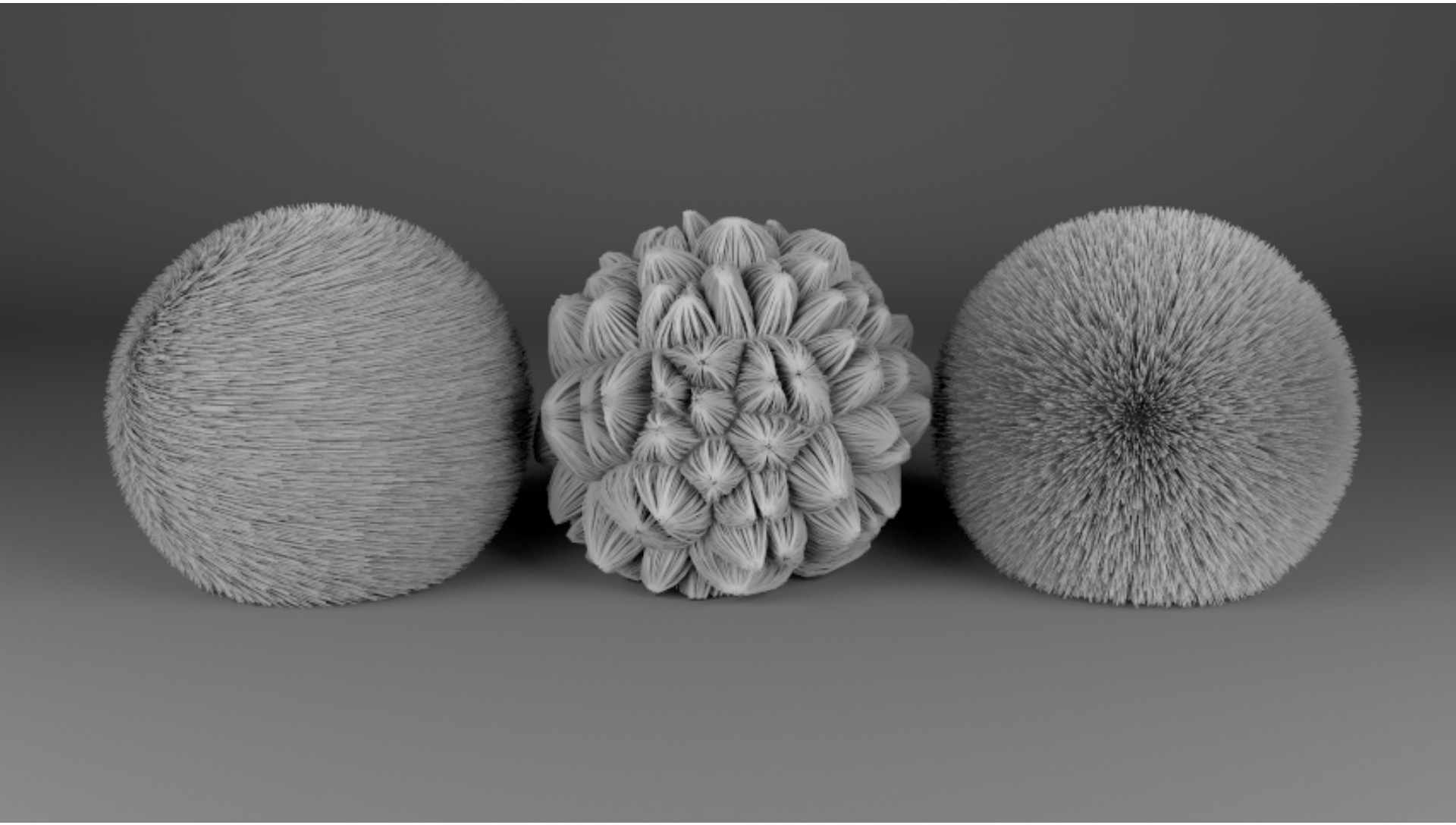
$$d = \mathbf{d} \cdot (\mathbf{e} - \mathbf{v}_0) \quad e = (\mathbf{v}_1 - \mathbf{v}_0) \cdot (\mathbf{e} - \mathbf{v}_0)$$

- We then clamp *s and t* in the appropriate ranges and check whether their distance is less then the segment radius

Ray-line intersection

```
bool intersect_line(ray3f ray, vec3f p0, vec3f p1,
  vec2f* uv, float* dist) {
  // setup intersection params
  auto u = ray.d; auto v = p1 - p0; auto w = ray.o - p0;
  // compute values to solve a linear system
  auto a = dot(u, u); auto b = dot(u, v); auto c   = dot(v, v);
  auto d = dot(u, w); auto e = dot(v, w); auto det = a*c - b*b;
  // check determinant and exit if lines are parallel
  if (det == 0) return false; // can use epsilon
  // compute distance on ray and segment, clamped to corners
  auto t = (b*e-c*d)/det; auto s = clamp((a*e-b*d)/det, 0, 1);
  // exit if not within bounds
  if (t < ray.tmin || t > ray.tmax) return false;
  // compute segment-segment distance on the closest points
  auto pr=ray.o+ray.d*t; auto pl=p0+(p1-p0)*s; auto prl=pr-pl;
  // check with the line radius at the same point
  auto d2 = dot(prl, prl); auto r  = r0 * (1 - s) + r1 * s;
  if (d2 > r * r) return false;
  // intersection occurred: set params and exit
  *uv = {s, sqrt(d2) / r}; *dist = t; return true;
}
```


Ray-line intersection



Interpolation in ray tracing

- When values are stored at vertices, use barycentric interpolation to define values across the whole surface that:
  - match the values at the vertices
  - are continuous across edges
  - are piecewise linear (linear over each triangle)
- How to compute interpolated values
  - during triangle intersection compute barycentric coords
  - use barycentric coords to average attributes given at vertices

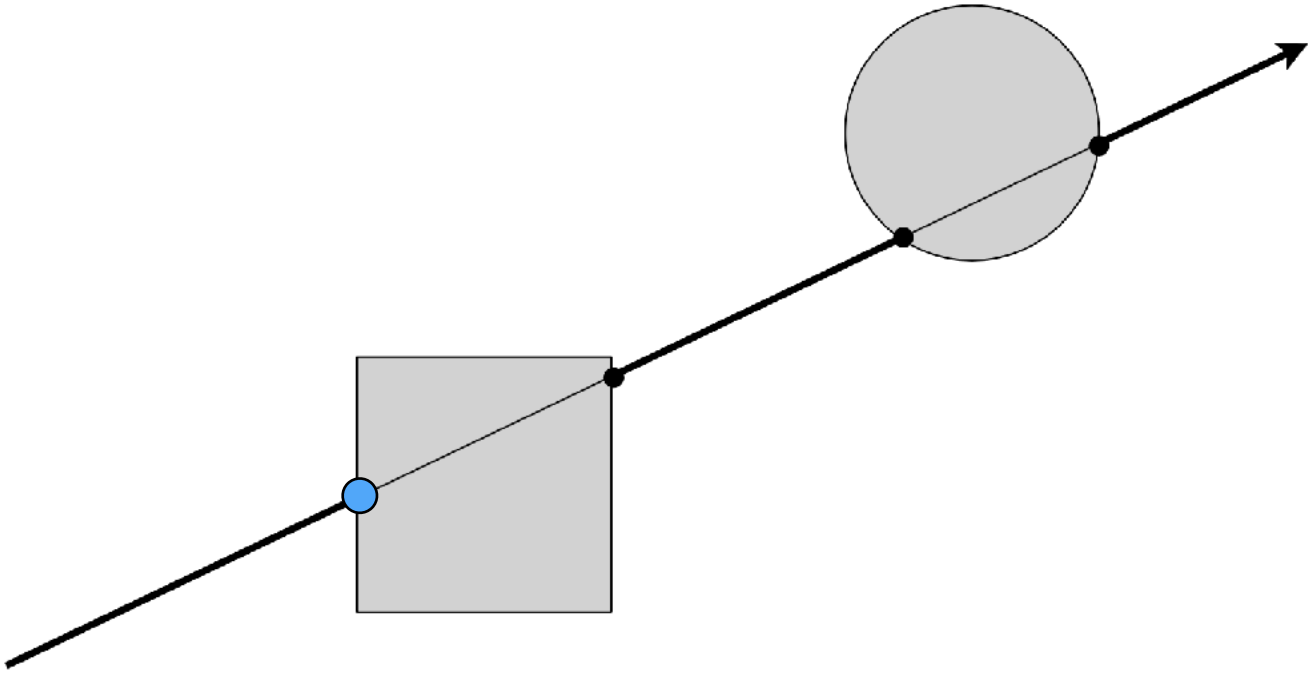
Ray intersection in software

- Option 1: All shapes intersectable via virtual functions
  - but virtual functions are slow
- Option 2: Focus on some shapes only, use external functions
  - much faster and common in real systems, e.g. Intel Embree
  - basic shapes: triangles, lines, points
  - extended shapes: curves, subdivision surfaces
  - returns whether we intersect, the ray distance and element barycentric coordinates

```
bool intersect_triangle(ray3f r, vec3f a, vec3f b,
  vec3f c, float* ray_t, vec2f* uv);
```


Ray-scene intersection

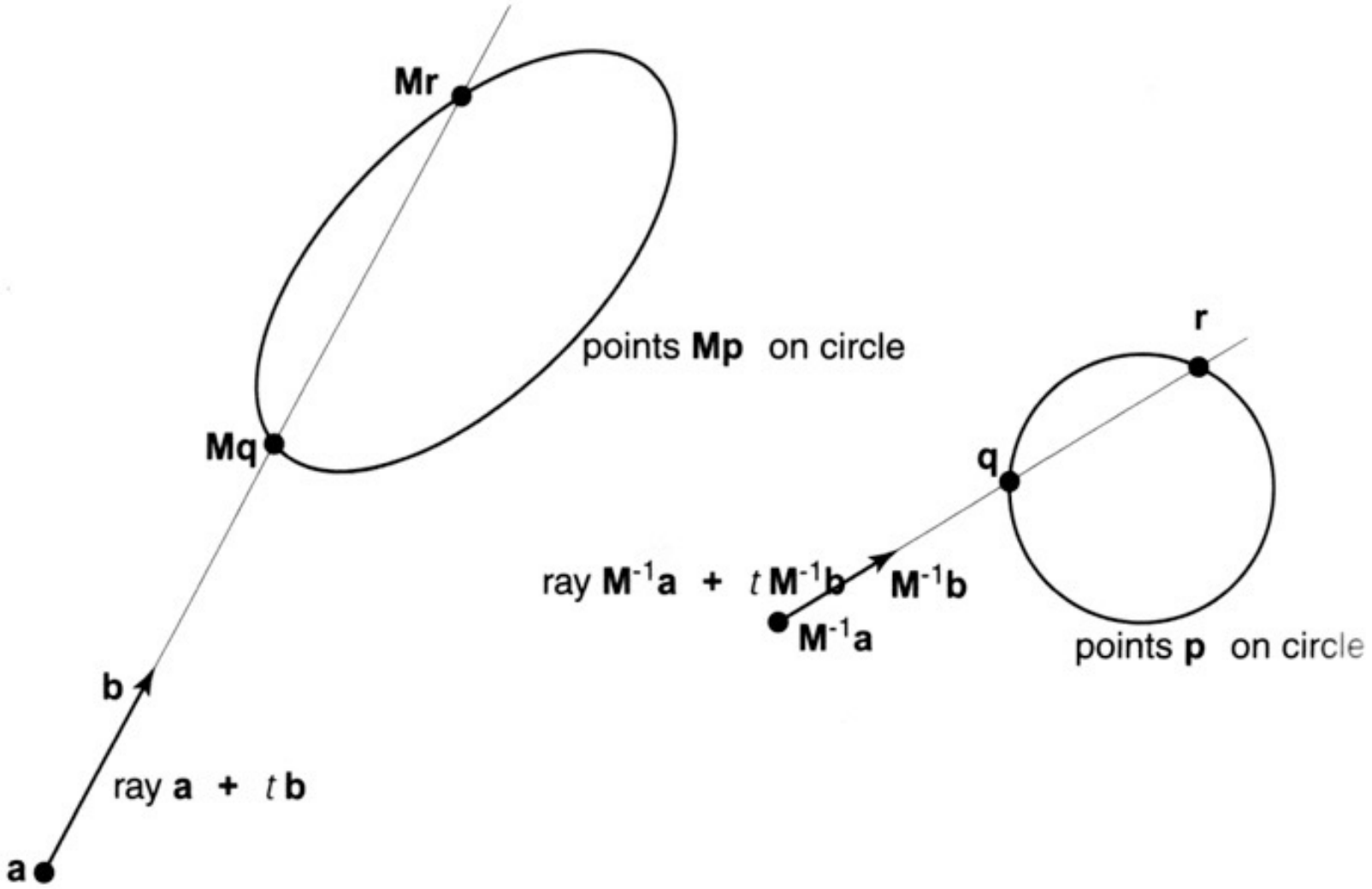
- Scenes have many instances: search for the *closest* intersection
- Loop over instances, ignoring those that don't intersect



Ray-instance intersection

- Each instance define its own coordinate system
  - mesh vertices are written w.r.t. a local coordinate system
- Rays are in world coordinates
- So we cannot intersect ray with triangle instances directly
- Option 1: transform each triangle to world coordinates
  - too expensive since we have many triangles
- Option 2: transform the ray in local coordinates
  - ray distance remains the same
  - ray origin and direction are transformed

Ray-instance intersection



Ray-scene intersection

- Return a record of the intersected instance, its shape element and the shape barycentric coordinates

```
struct intersection3f {
    float distance; // ray distance
    int object;     // object id
    int element;    // element id (triangle, line, point)
    vec2f uv;       // barycentric coordinates
};
```


Ray-scene intersection

- Convenient implementation: keep track of the closest shape by updating the *tmax* value with the closest one
  - this way we skip all further shapes with the shape checks

```
bool intersect(scene* scene, ray3f ray,
    intersection3f* isec) {
  auto hit = false;
  for(auto object_id : range(scene->objects.size())) {
    auto object = scene->objects[object_id];
    auto tray = transform_ray(inverse(object->frame), ray);
    if(!intersect(object->shape, tray, isec)) continue;
    hit = true;
    isec->object = object_id;
    ray.tmax = isec->ray_t;
  }
  return hit;
}
```


Ray-shape intersection

- Same as scene, but no need to transform ray
- Example here for triangle mesh

```
bool intersect(shape* shape, ray3f ray,
    intersection3f* isec) {
  auto hit = false;
  for(auto element_id : range(shape->triangles.size())) {
    auto t = shape->triangles[element_id];
    if(!intersect_triangle(ray, shape->pos[t.x],
        shape->pos[t.y], shape->pos[t.z],
        isec->distance, isec->uv)) continue;
    hit = true;
    isec->element = element_id;
    ray.tmax = isec->distance;
  }
  return hit;
}
```

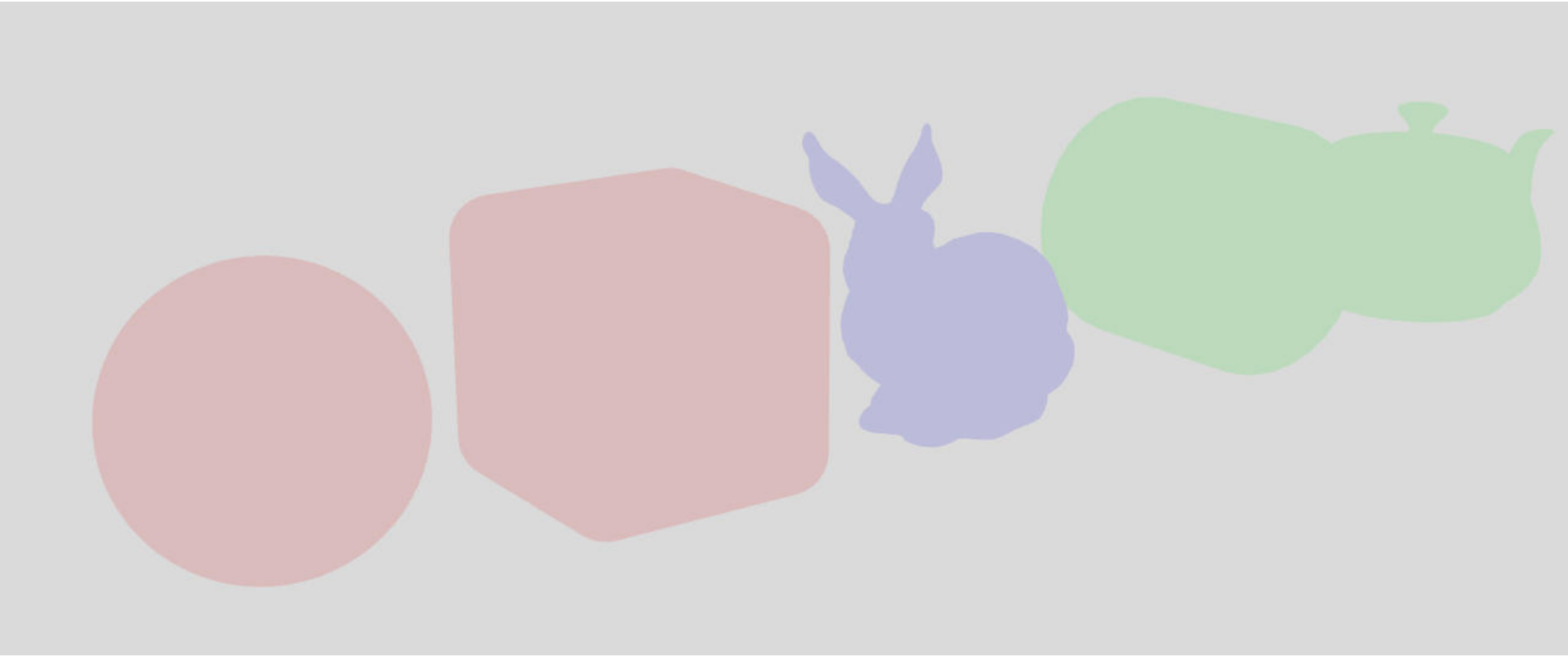

Ray-scene intersection

- Image with eye ray generation and scene intersection

```
for(auto j : range(height)) {
  for(auto i : range(width)) {
    auto (u, v) = pixel_uv(i,j);
    auto ray = camera_ray(u,v);
    img[i,j] = shade(scene, ray);
  }
}

vec3f shade_color(scene* scene, ray3f ray) {
  auto isec = intersection3f{};
  if(!intersect_scene(scene, ray, &isec))
    return zero3f;
  auto object = scn->objects[isec.object];
  return object->material->color;
}
```


Ray-scene intersection

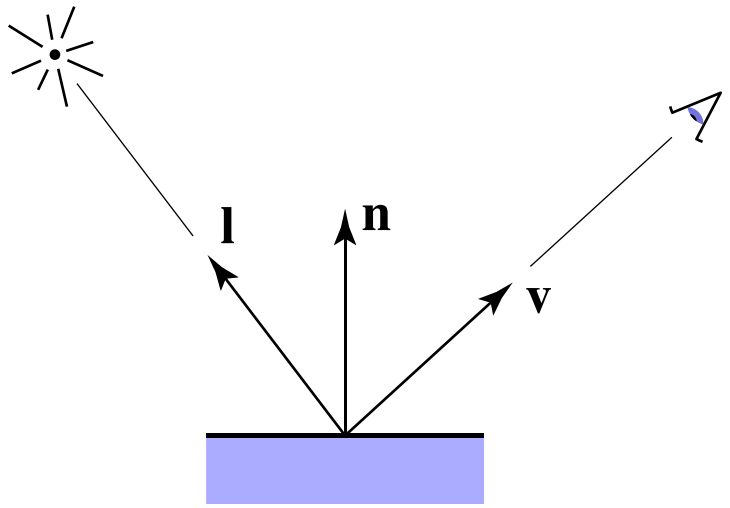
- Image with eye ray generation and scene intersection



Shading

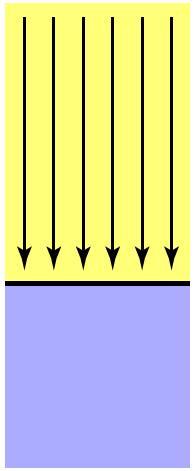
Shading

- Compute light reflected toward camera
- Inputs:
  - eye direction
  - light direction
    (for each of many lights)
  - surface normal
  - surface parameters
    (color, shininess, …)

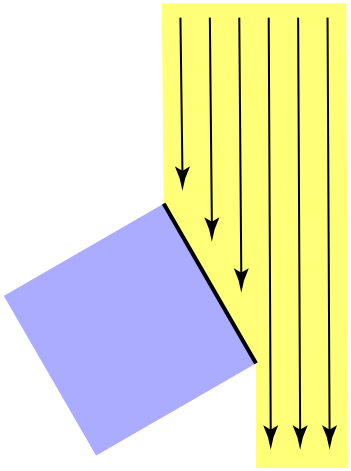


Diffuse reflection

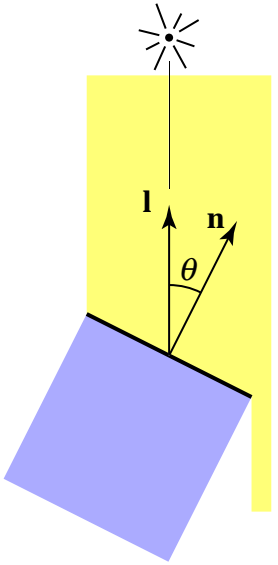
- Light is scattered uniformly in all directions
  - the surface color is the same for all viewing directions
- Lambert's cosine law



Top face of cube receives a certain amount of light

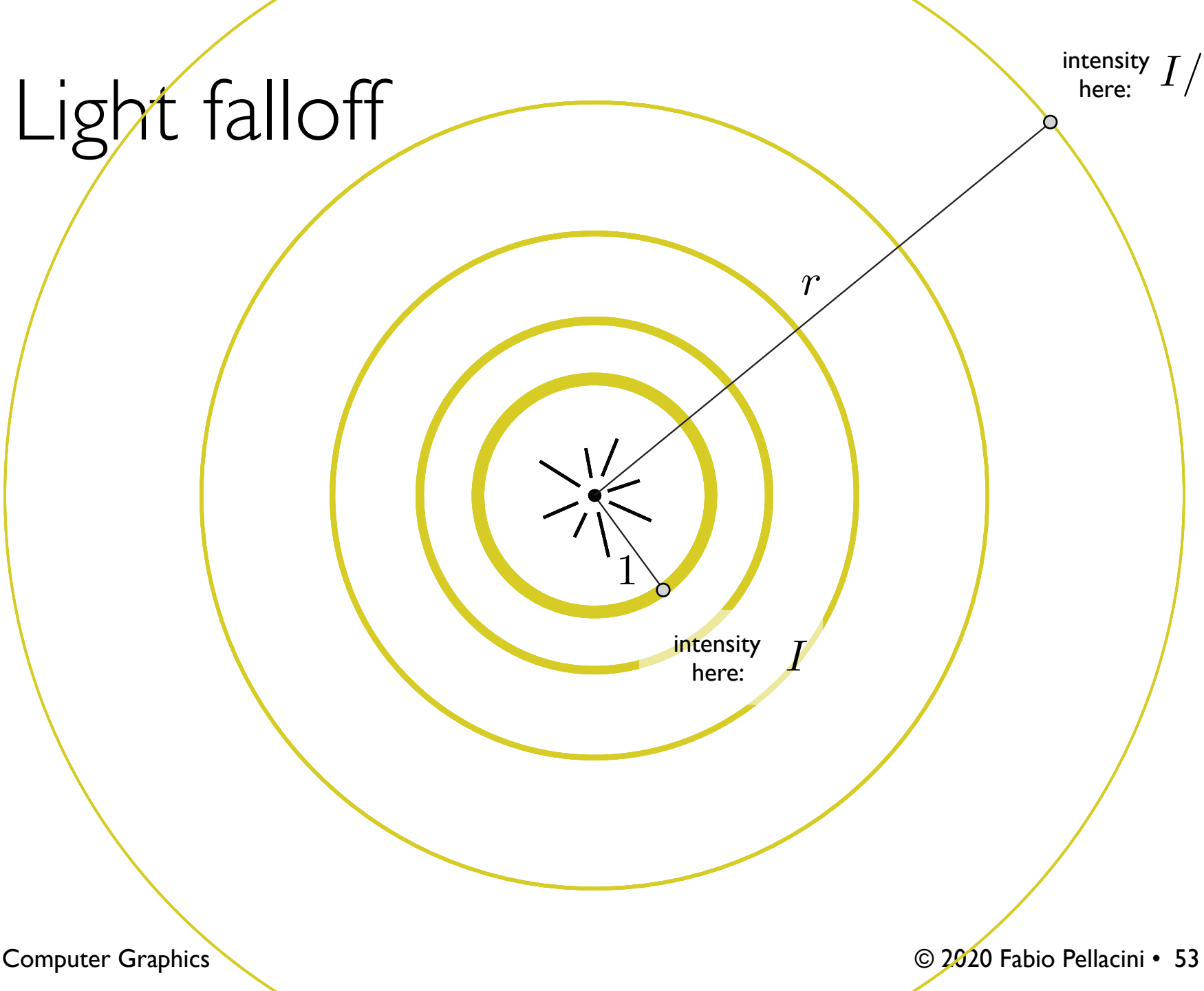


Top face of 60° rotated cube intercepts half the light



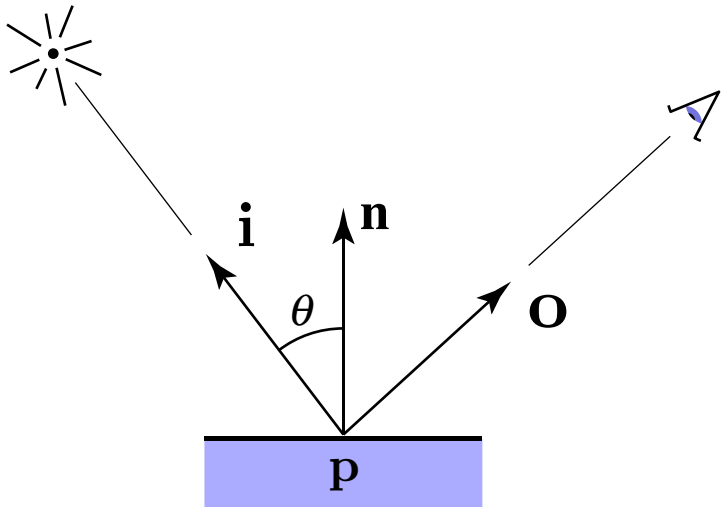
In general, light per unit area is proportional to cos θ = l • n

Light falloff



Diffuse illumination

- Shading depends on lighting direction and normal
- Normalize by pi as discussed later



$$L_d = \frac{k_d}{\pi} \frac{I}{|\mathbf{s}-\mathbf{p}|^2} max(0, \mathbf{n} \cdot \mathbf{i})$$

$$\mathbf{i} = \frac{\mathbf{s}-\mathbf{p}}{|\mathbf{s}-\mathbf{p}|}$$

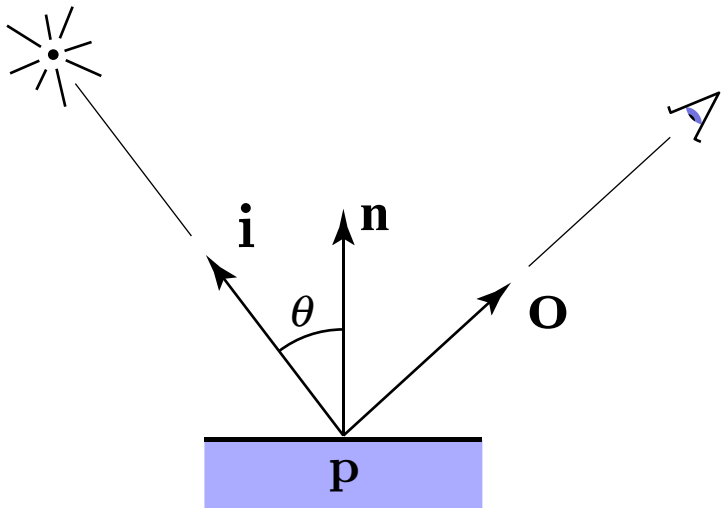
illumination from source

diffuse coefficient

diffusely reflected illumination

Diffuse illumination

- Take the absolute value of the dot product to ignore sidedness
- Sum contribution of multiple lights



$$L_d = \sum_i \frac{k_d}{\pi} \frac{I_i}{|\mathbf{s}_i - \mathbf{p}|^2} |\mathbf{n} \cdot \mathbf{i}_i|$$

sum over lights

ignore sideness

Computing shape values

- First, evaluate normal and position from intersection

```
T interpolate_triangle(T a, T b, T c, vec2f uv) {
  return a * (1 - uv.x - uv.y) + b * uv.x + c * uv.y;
}

vec3f eval_normal(shape* shape, int element, vec2f uv) {
  auto t = triangles[element];
  return normalize(interpolate_triangle(
    shape->normals[t.x], shape->normals[t.y],
    shape->normals[t.z], uv));
}

vec3f eval_position(shape* shape, int element, vec2f uv) {
  auto t = triangles[element];
  return interpolate_triangle(shape->positions[t.x],
    shape->positions[t.y], shape->positions[t.z], uv);
}
```

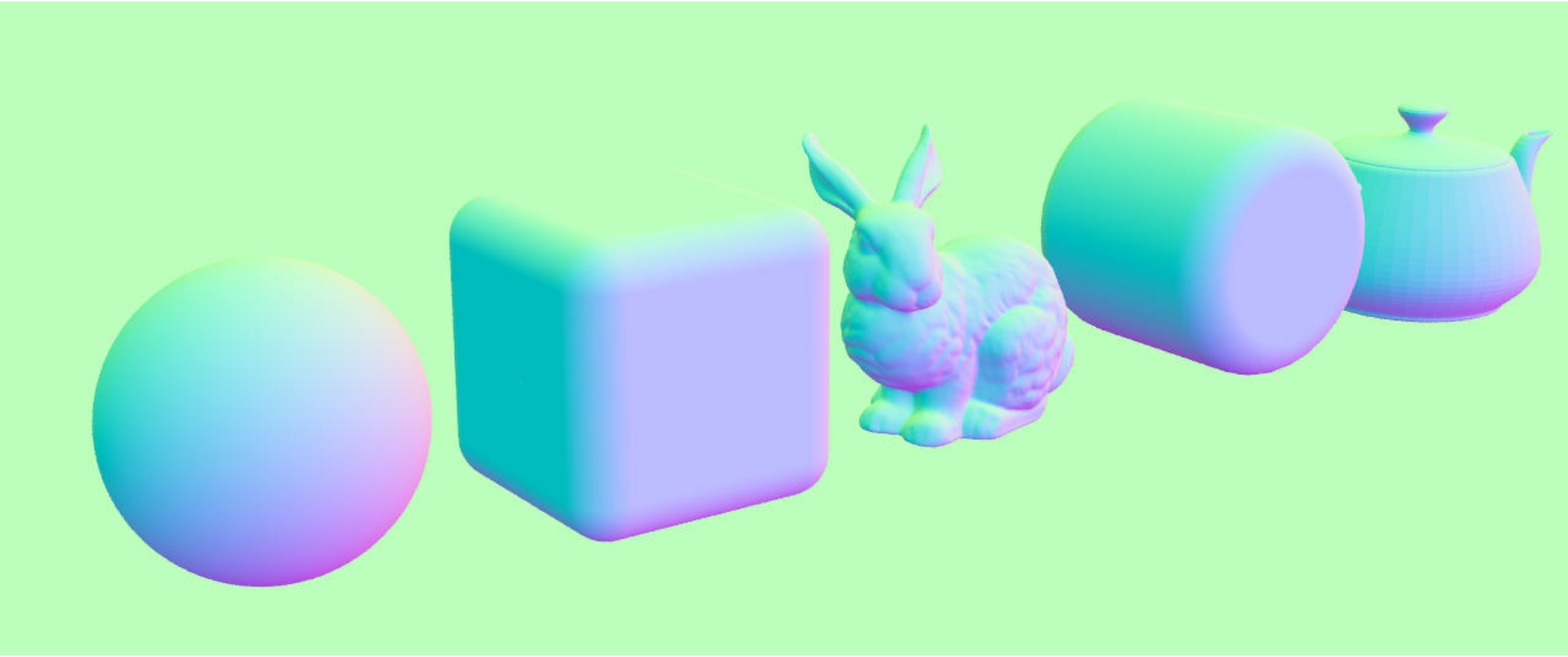

Normal shading

- Can make a small preview

```
vec3f shade_normal(scene* scene, ray3f ray) {
  auto isec = intersection3f{};
  if(!intersect_scene(scene, ray, &isec))
    return zero3f;
  auto object = scn->objects[isec.object];
  auto normal = transform_direction(object->frame,
    evalute_normal(object->shape, isec.element, isec.uv));
  return normal * 0.5 + 0.5;
}
```


Normal shading

- Can make a small preview



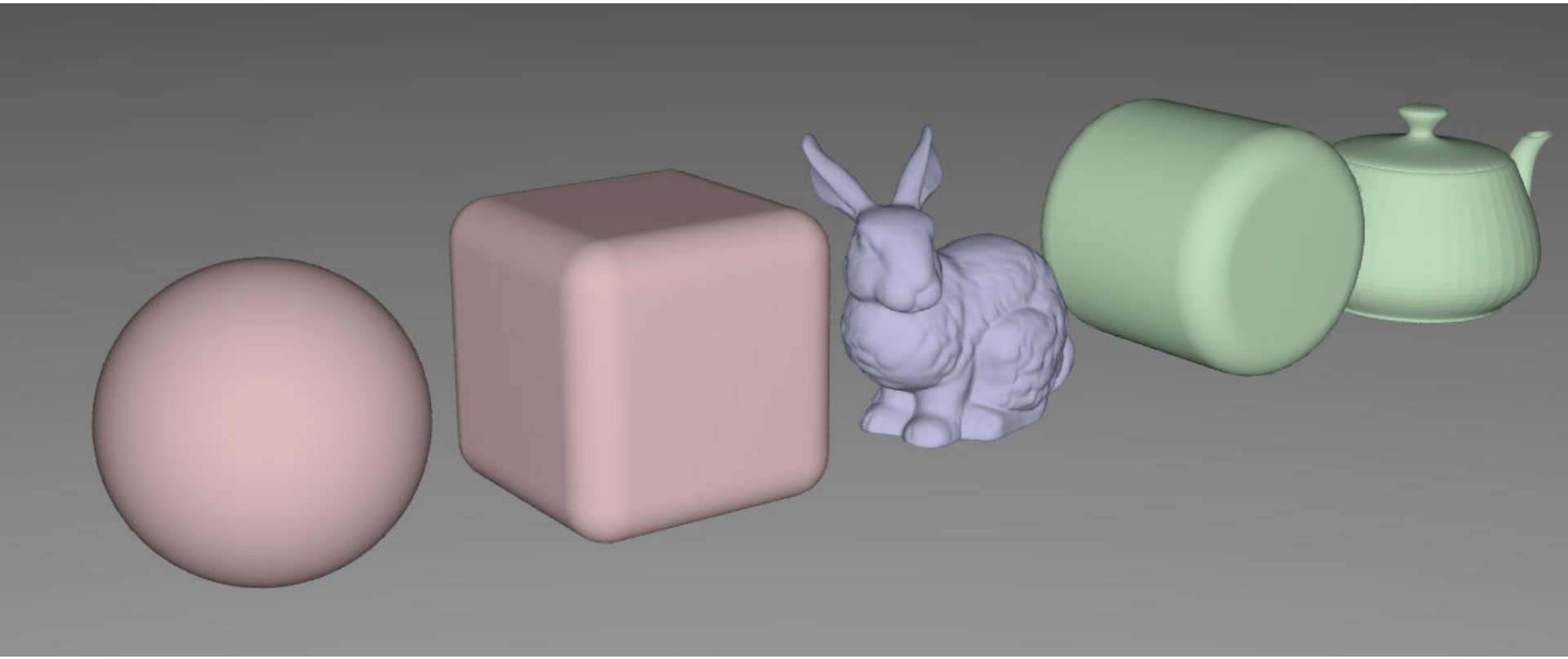
Diffuse illumination

- Then, make a simple preview by pretending there is a light at the camera position, and avoid light falloff and normalization

```
vec3f shade_eyelight(scene* scene, ray3f ray) {
  auto isec = intersection3f{};
  if(!intersect_scene(scene, ray, &isec))
    return zero3f;
  auto object = scn->objects[isec.object];
  auto normal = transform_direction(object->frame,
    evalute_normal(object->shape, isec.element, isec.uv));
  return object->material->color * dot(normal, -ray.d);
}
```

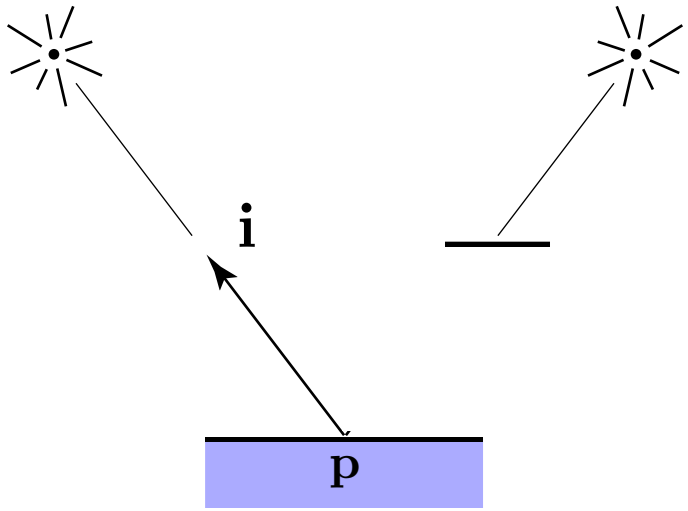

Diffuse illumination

- Then, make a simple preview by pretending there is a list at the camera position, and avoid light falloff



Shadows

- Surface is only illuminated if nothing blocks its view of the light
- Check for that by casting a ray toward the light



visibility of source

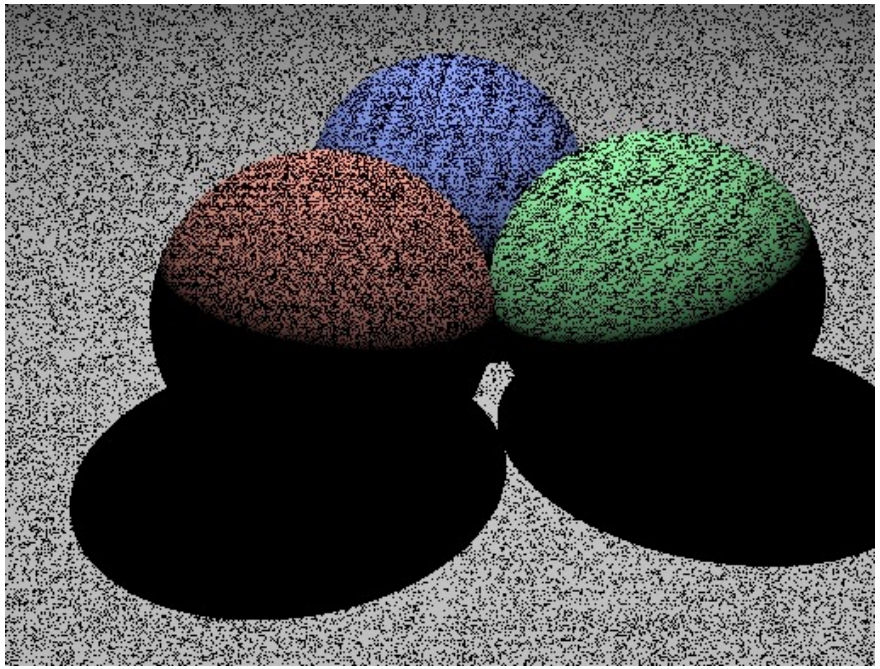
$$L_d = \sum_i \frac{k_d}{\pi} \frac{I_i}{|\mathbf{s}_i - \mathbf{p}|^2} V(\mathbf{s}_i, \mathbf{p}) |\mathbf{n} \cdot \mathbf{i}_i|$$

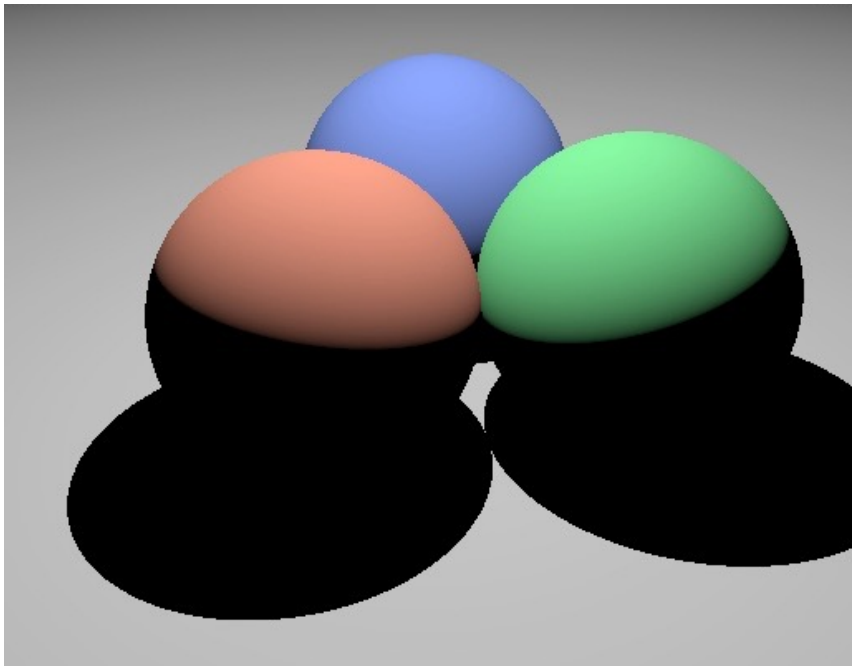
$$V(\mathbf{s}, \mathbf{p}) = !\text{intersect}(ray(\mathbf{p}, \mathbf{i}, \epsilon, |\mathbf{p} - \mathbf{s}|))$$

Shadow rounding errors

- The ray may intersect the shape at the shading point
- Add a small epsilon to avoid this

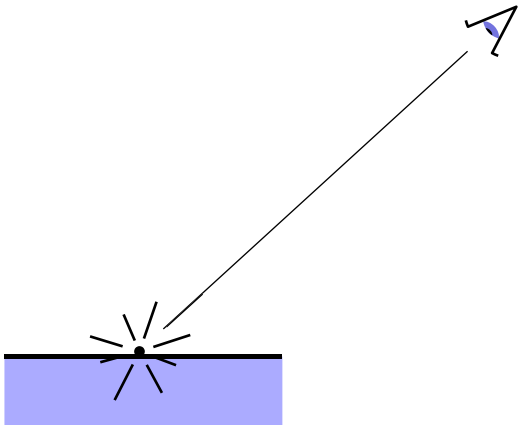
```
auto sr = ray3f{p,l,eps,r};
```





Emission

- Surface not only reflect light, but also emit it
- We can simply add a constant term for this



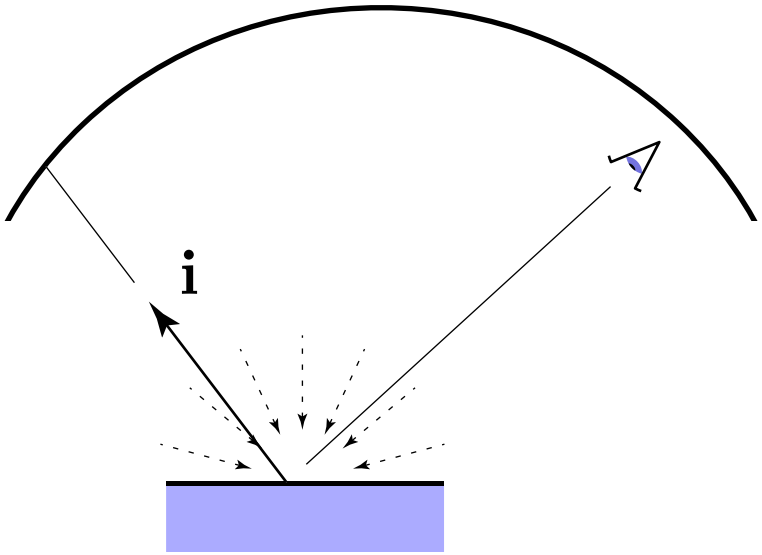
$$L_e = k_e$$

emitted light

emission coefficient

Environment illumination

- Illumination from the environment comes from all directions
- We approximate it by averaging the illumination of random directions
- More directions leads to better approximation



size of hemisphere

environment illumination

visibility of source

$$L_d \approx \frac{2\pi}{n} \sum_{i}^{n} \frac{k_d}{\pi} E_e(\mathbf{i}_i) V(\mathbf{p}, \mathbf{i}_i) |\mathbf{n} \cdot \mathbf{i}_i|$$

reflected illumination

average of *n* samples

Picking random directions

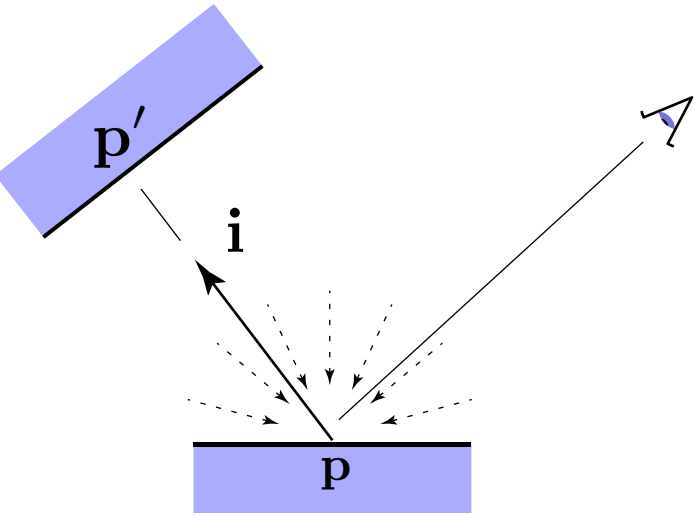
- To generate directions over the hemisphere, we pick two random numbers and use them as the cylindrical coordinates of a direction
- Note the formula is given in local coordinates, so transform it to world with a basis generate from the normal

$$\mathbf{i}_l = \left(\sin(2\pi r_1)\sqrt{1-r_2^2}, \cos(2\pi r_1)\sqrt{1-r_2^2}, r_2 \right)$$

$$\mathbf{i} = F_{\mathbf{n}} \cdot \mathbf{i}_l$$

Indirect illumination

- Illumination comes also from all other objects
- As before, we can average the contribution from random directions
- We can determine the closely points via ray casting



emitted and reflected illumination from other points

$$\mathbf{p}'_i = \text{intersect}(ray(\mathbf{p}, \mathbf{i}_i))$$

$$L(\mathbf{p}) \approx L_e(\mathbf{p}) + \frac{2\pi}{n} \sum_i^n \frac{k_d}{\pi} L(\mathbf{p}'_i) |\mathbf{n} \cdot \mathbf{i}_i|$$

emitted and reflected illumination

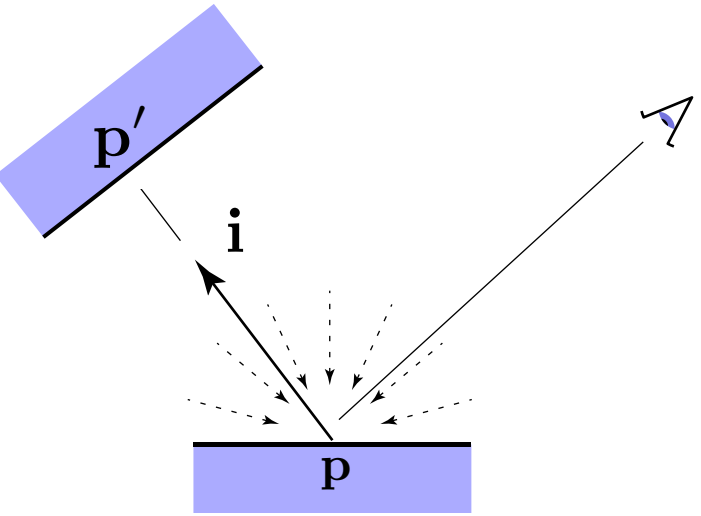
emitted illumination

size of the hemisphere over samples

reflected illumination

Indirect illumination

- Note how the formulation is recursive; a reasonable first approximation is to stop after 4-8 bounces
- Also, the number samples grows exponentially; we can simply pick only one sample and take the average in the "outer loop" shown later
- Finally, we can include environment illumination too by simply returning if if the ray does not hit a surface



$$L(\mathbf{p}) \approx L_e(\mathbf{p}) + \frac{2\pi}{n} \sum_{i}^{n} \frac{k_d}{\pi} L(\mathbf{p}'_i) |\mathbf{n} \cdot \mathbf{i}_i|$$

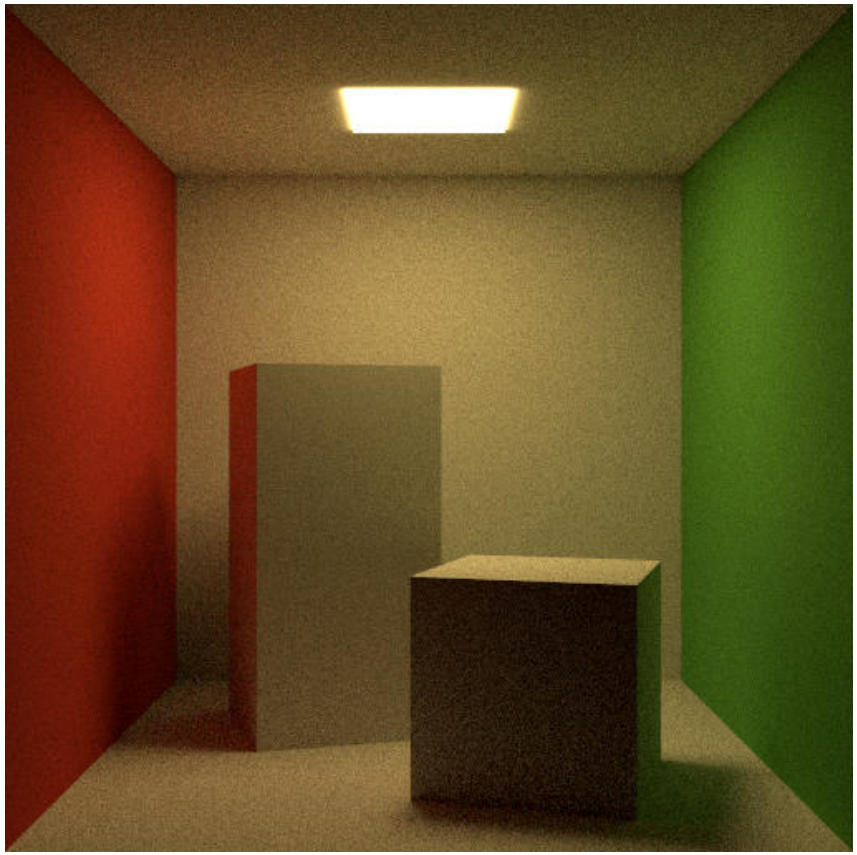
emitted and reflected illumination

emitted illumination

reflected illumination

Indirect illumination

- Here is the effect of indirect illumination with an emissive surface



Indirect illumination

- Here is the effect of indirect illumination with an emissive surface

Illumination approximation

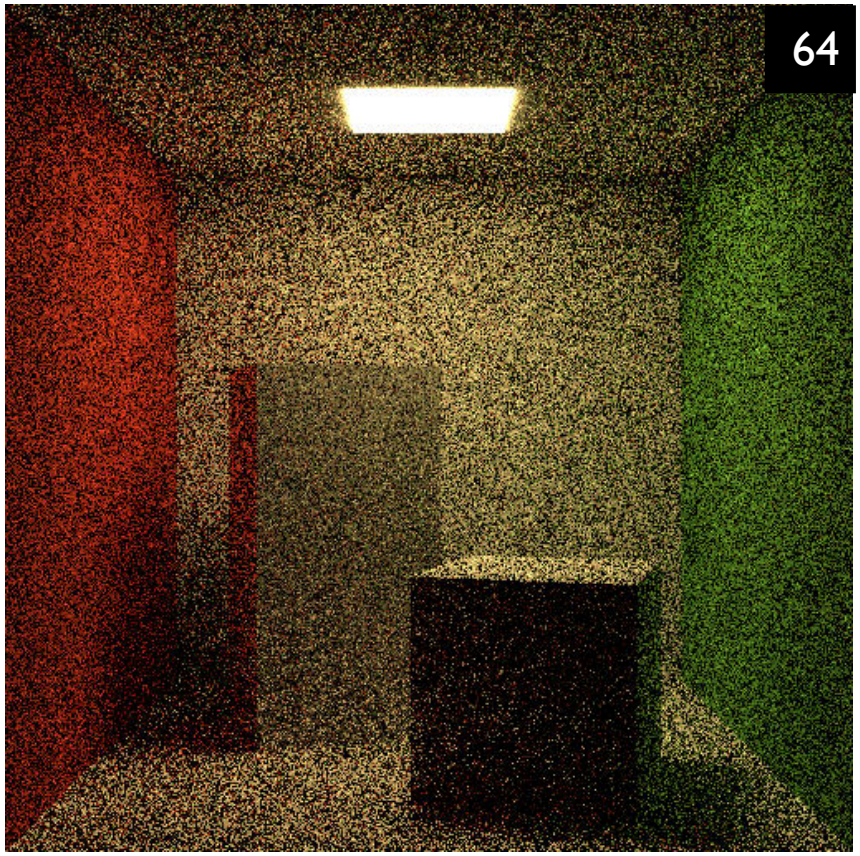
- Errors show up as noise: more samples means less noise

Illumination approximation

- Errors show up as noise: more samples means less noise

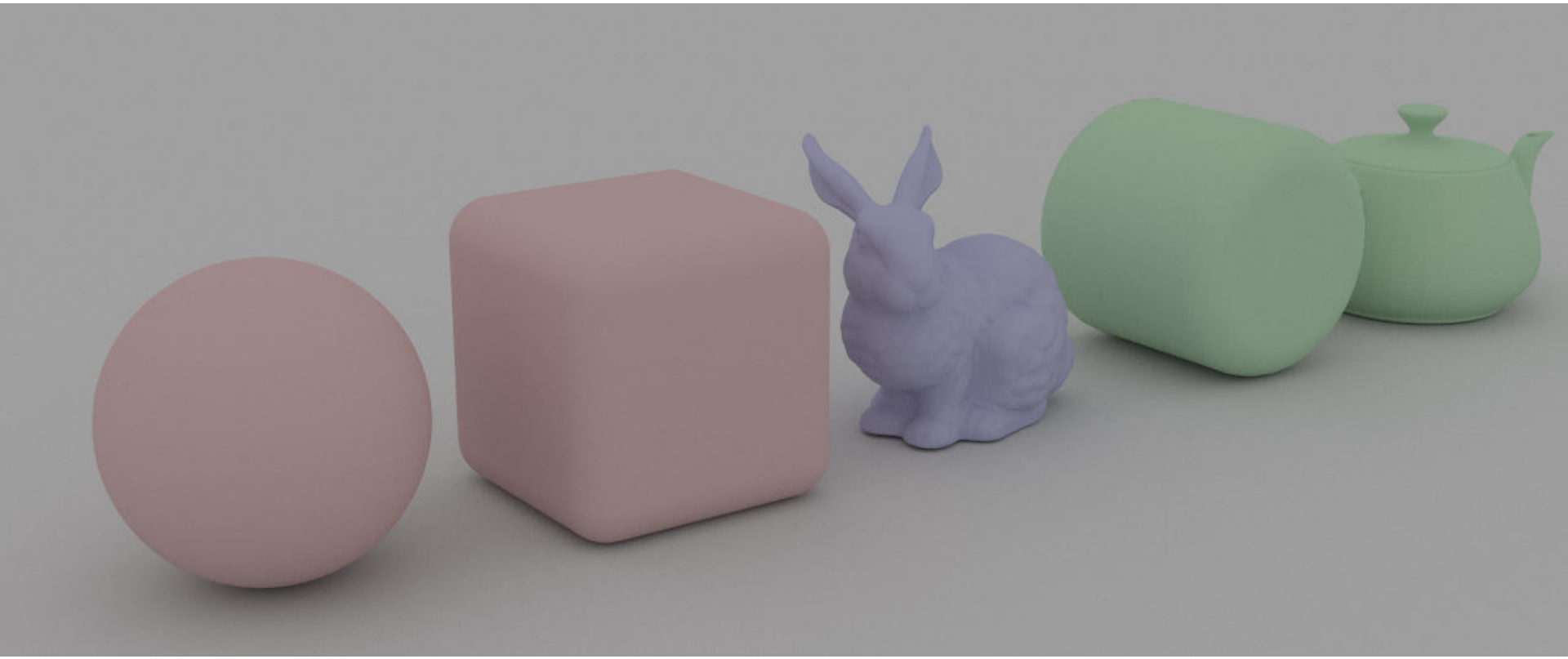
Illumination approximation

- Errors show up as noise: more samples means less noise



Indirect and environment

- Shown here is the effect of indirect illumination with an environment

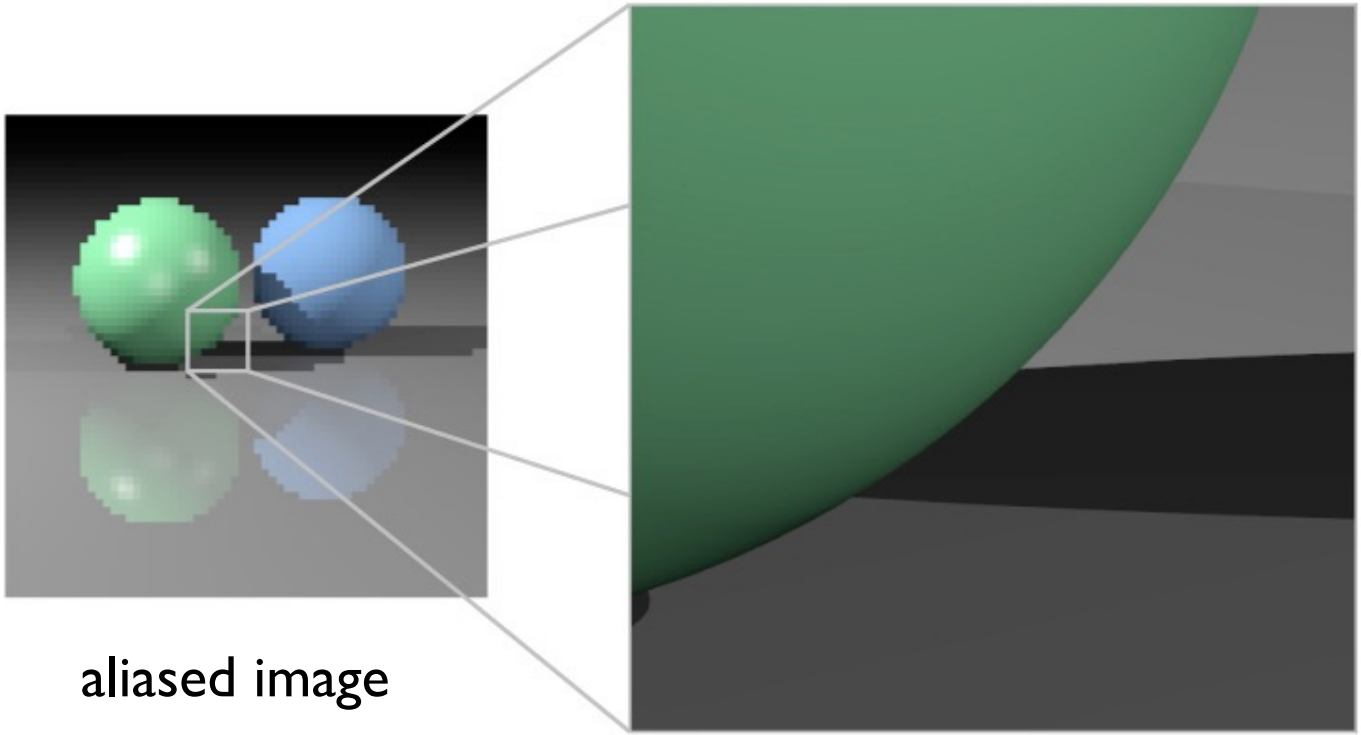


Indirect illumination

```
vec3f shade_indirect(scene* scene, ray3f ray, int bounce) {
  auto isec = intersection3f{};
  if(!intersect_scene(scene, ray, &isec))
    return eval_environment(scene, ray.d);
  auto object = scn->objects[isec.object];
  auto normal = transform_direction(object->frame,
    evalute_normal(object->shape, isec.element, isec.uv));
  auto position = transform_point(object->frame,
    evalute_position(object->shape,isec.element, isec.uv));
  auto radiance = object->material->emission;
  if(bounce >= max_bounce) return radiance;
  auto incoming = sample_hemisphere(normal, rand2f(rng));
  radiance += (2 * pi) * object->material->color / pi *
    shade_indirect(scene, ray3f{position, incoming},
        bounce+1) * dot(normal, incoming);
  return radiance;
}
```

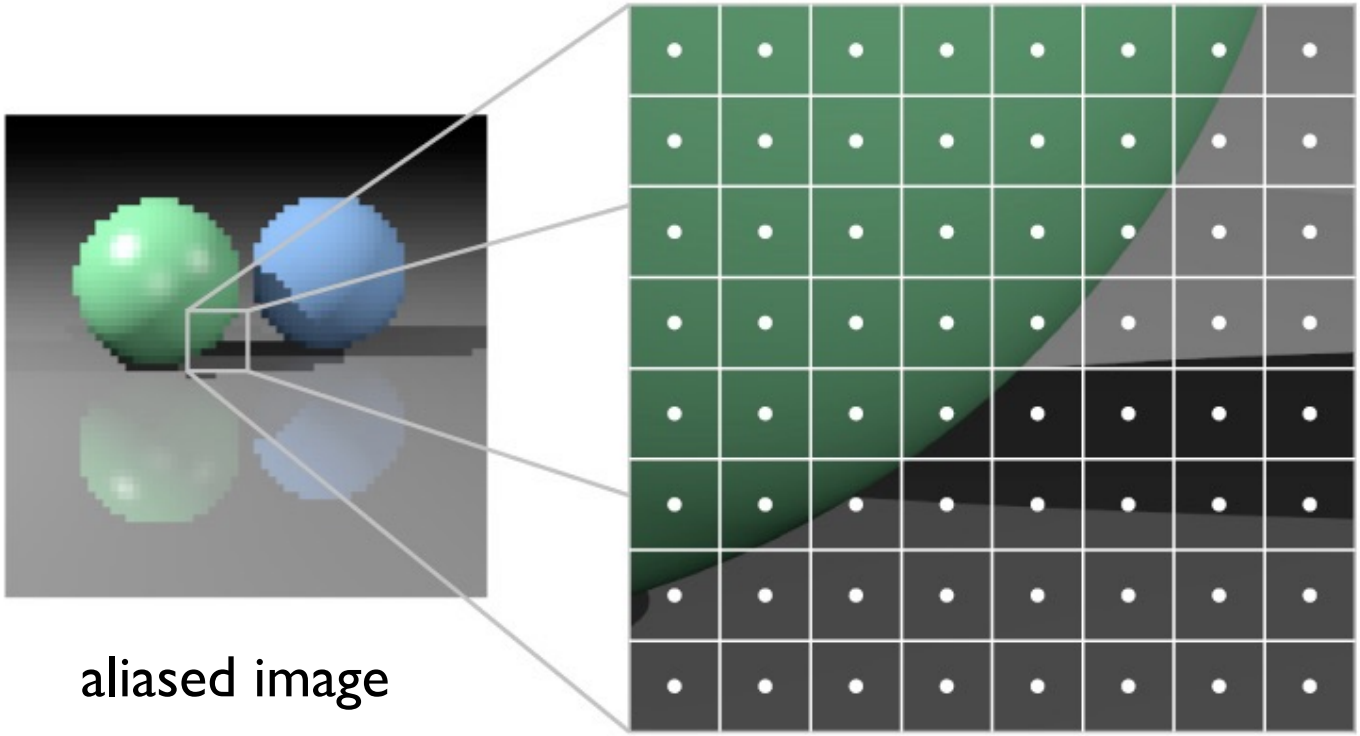

Antialiasing

Antialiasing in ray tracing



aliased image

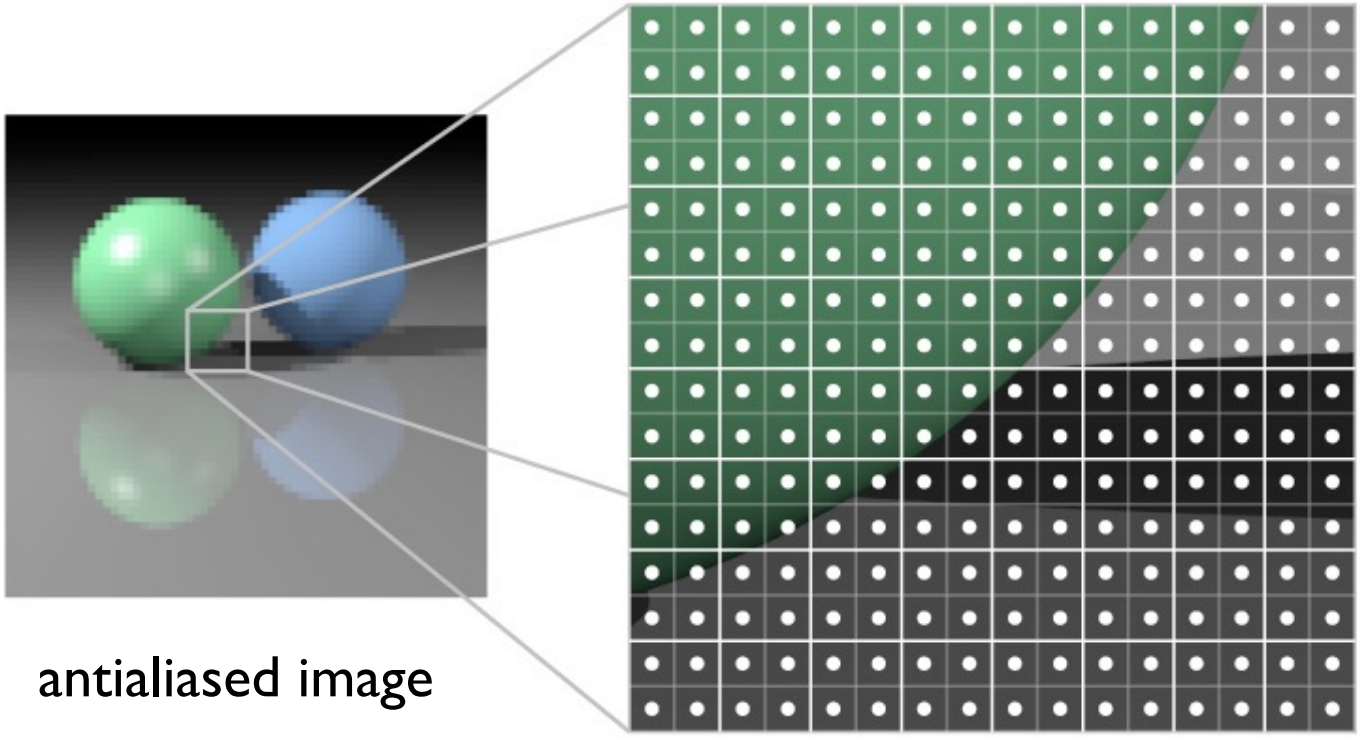
Antialiasing in ray tracing



aliased image

one sample per pixel

Antialiasing in ray tracing



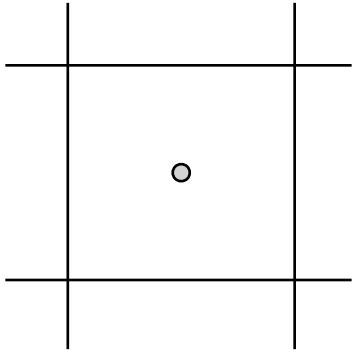
antialiased image

four samples per pixel

Aliased raytracing

- Using only one sample per pixel introduces aliasing

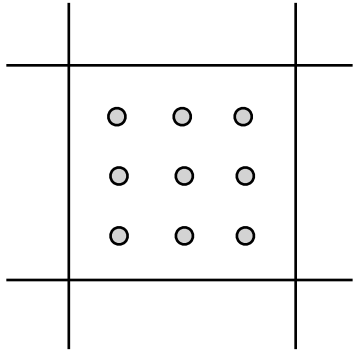
```
// aliased raytracing
for(auto j : range(img.height)) {
  for(auto i : range(img.width)) {
    auto uv = (vec2f{i,j} + 0.5f) / img.size;
    auto ray = camera_ray(cam, uv);
    img[i,j] = shade(scene, ray);
  }
}
```



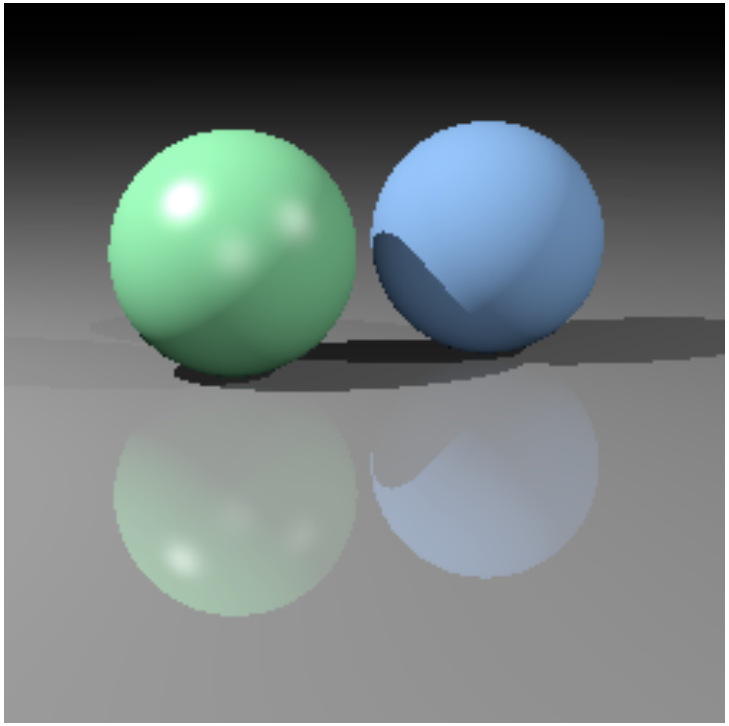
Antialiasing by supersampling

- Average over multiple points in each pixel to reduce aliasing

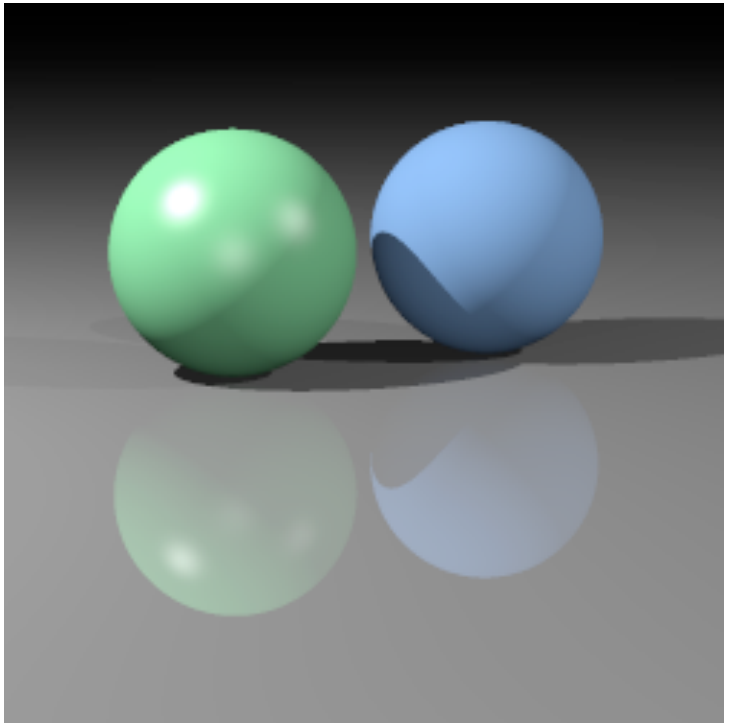
```
// antialiased with ns^2 samples per pixel
for(auto j : range(img.height)) {
  for(auto i : range(img.width)) {
    img[i,j] = {0,0,0};
    for(auto sj : range(ns)) {
      for(auto si : range(ns)) {
        auto puv = (vec2f{si,sj}+0.5f)/ns;
        auto uv = (vec2f{i,j} + puv) / img.size;
        auto ray = camera_ray(cam, uv);
        img[i,j] += shade(scene, ray);
      }
    }
    img[i,j] /= ns*ns;
  }
}
```



Antialiasing in ray tracing



one sample per pixel

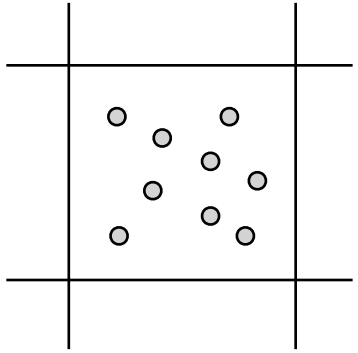


nine samples per pixel

Antialiasing by random sampling

- We can also take locations at random in the pixel
- This allows to take any number of samples and can serve as outer loop for the shading functions given before

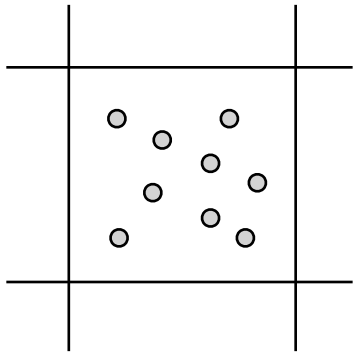
```
// antialiased with ns samples per pixel
auto rng = make_rng();
for(auto j : range(img.height)) {
  for(auto i : range(img.width)) {
    img[i,j] = {0,0,0};
    for(auto s : range(ns)) {
      auto puv = rand2f(rng);
      auto uv = (vec2f{i,j} + puv) / img.size;
      auto ray = camera_ray(cam, uv);
      img[i,j] += shade(scene, ray, rng);
    }
   img[i,j] /= ns;
  }
}
```



Progressive rendering

- By inverting the loops, we can view images as they are computed
- For simplicity, we keep an accumulated image and rngs per pixels

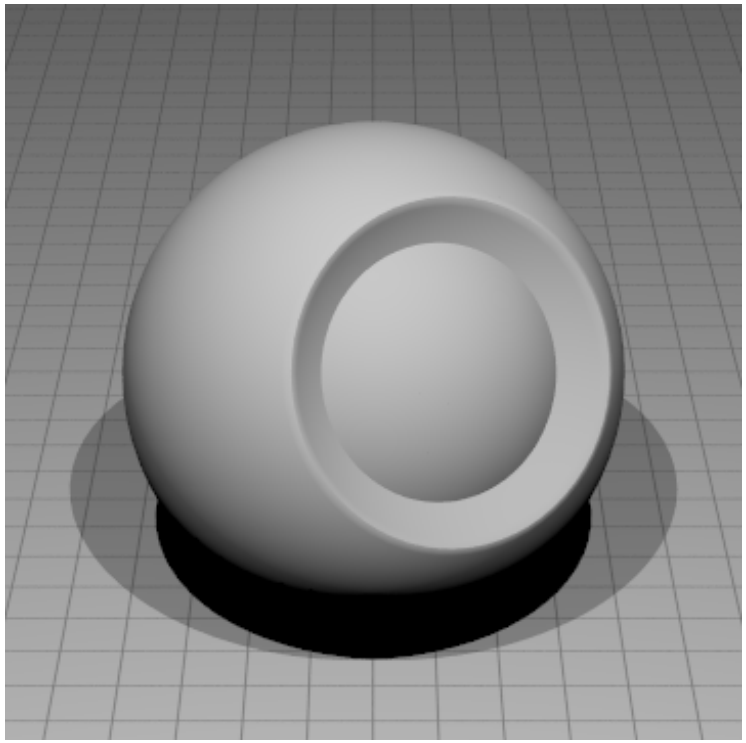
```
// progressive with ns samples per pixel
auto rngs = make_rngs();
auto acc = image{img.size, zero3f};
for(auto s : range(ns)) {
  for(auto j : range(img.height)) {
    for(auto i : range(img.width)) {
     auto puv = rand2f(rngs[i,j]);
      auto uv = (vec2f{i,j} + puv) / img.size;
      auto ray = camera_ray(cam, uv);
      acc[i,j] += shade(scene, ray, rngs[i,j]);
      img[i,j] = acc[i,j] / (s + 1);
    }
  }
  // can view or save image here
}
```

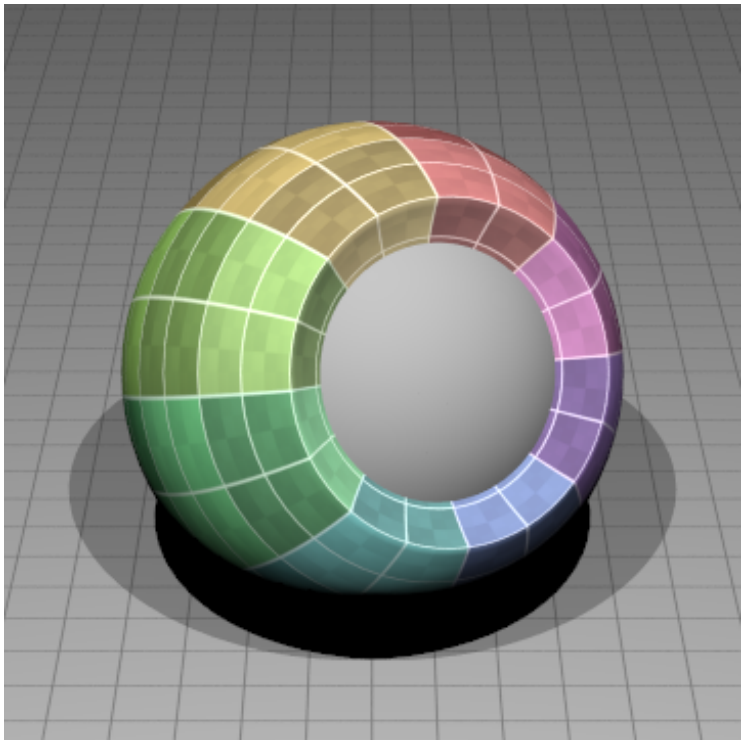


Textures

Textures

- Render textures by modifying material parameters *kd* and *ks* in the lighting equation
  - typically by multiplying the texture colors by the coefficients

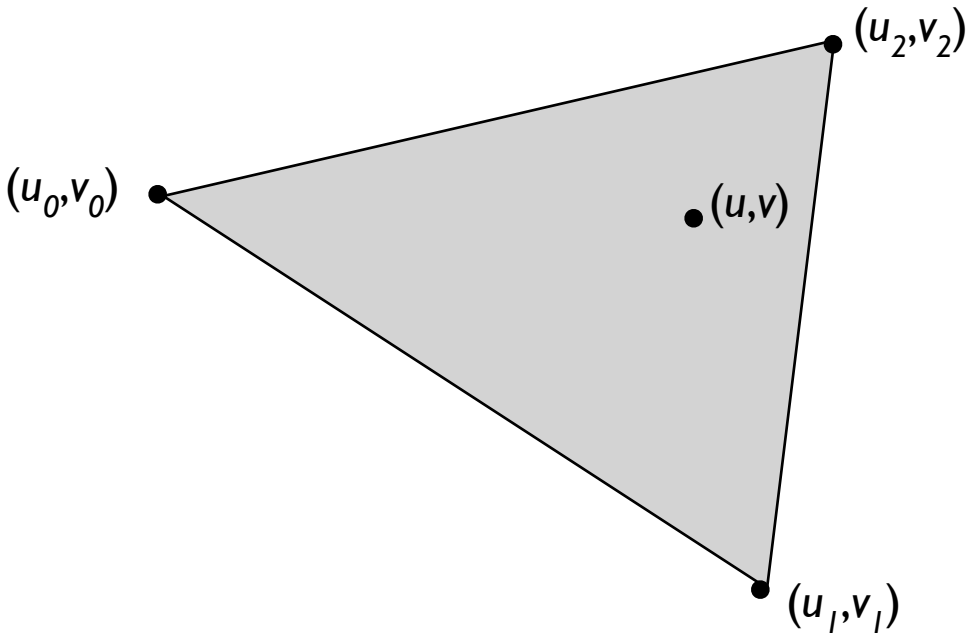




Textures

- First, compute the texture coordinates *(u,v)* of the intersection point with barycentric interpolation of the intersected triangle

$$u = \sum w_i u_i \quad v = \sum w_i v_i$$

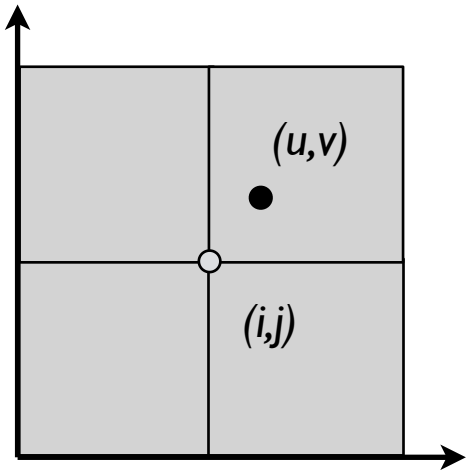


Lookup Texture Values

- Then compute the image coordinate *(i,j)* for those *(u,v)*, taking into account tiling

$$s = (u \bmod 1) \cdot width \quad t = (v \bmod 1) \cdot height$$

$$i = \text{floor}(s) \quad j = \text{floor}(t)$$



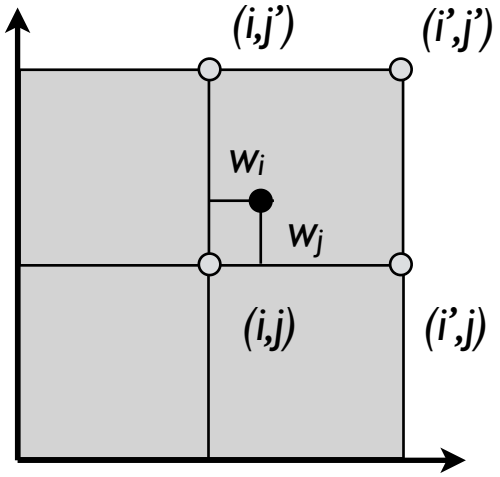
Lookup Texture Values

- Compute final value by interpolating between the closest 4 pixels; this avoids pixelation when we magnify the image

$$i' = (i+1) \bmod width \quad j' = (j+1) \bmod height$$

$$w_i = s - i \quad w_j = t - j$$

$$\mathbf{c} = (1-w_i)(1-w_j)\mathbf{c}_{ij} + w_i(1-w_j)\mathbf{c}_{i'j} + (1-w_i)w_j\mathbf{c}_{ij'} + w_i w_j \mathbf{c}_{i'j'}$$

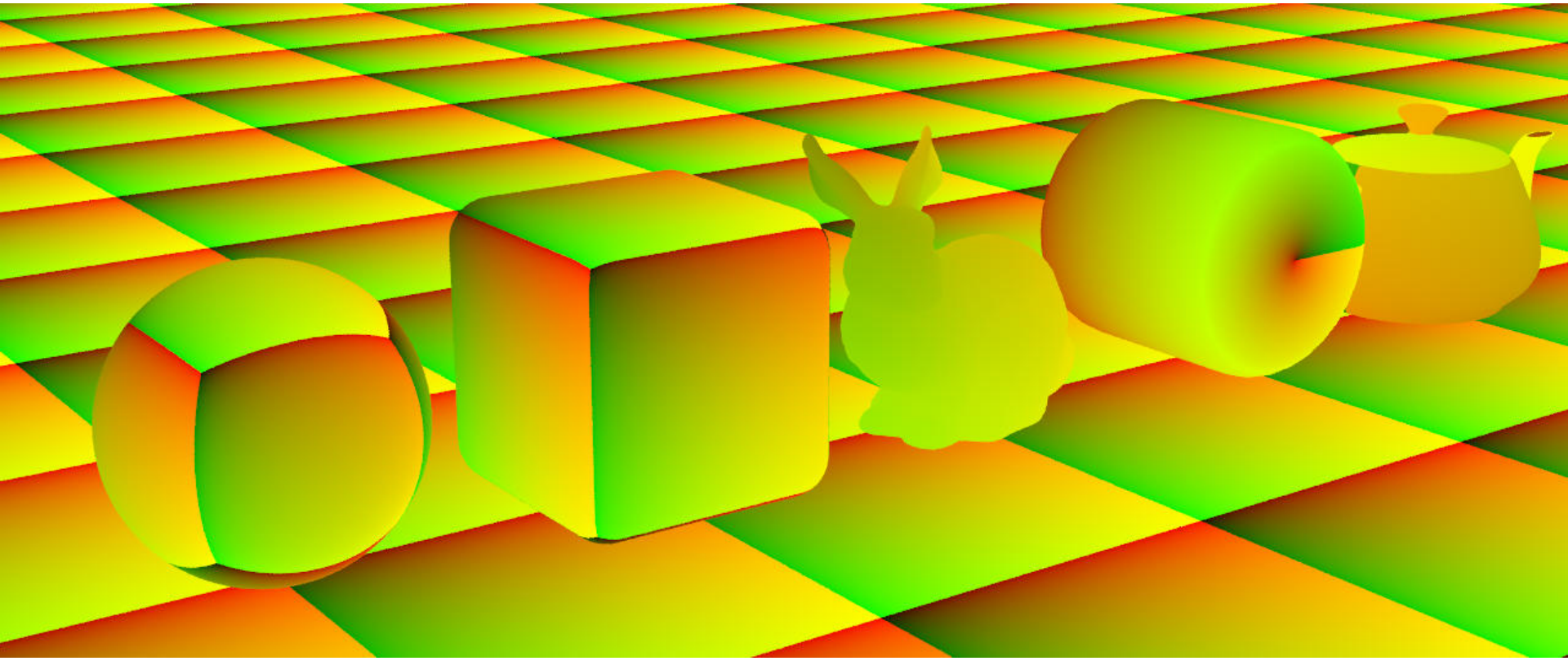


Lookup Texture Values

```
vec3f lookup(texture* texture, vec2i ij, bool no_srgb) {
  if(!texture->hdr.empty()) return texture->hdr[i,j];
  else if(no_srgb) return byte_to_float(texture->ldr[i,j])
  else return rgb_to_srgb(byte_to_float(texture->ldr[i,j]));
}

vec3f eval_texture(texture* texture, vec2f uv, bool no_srgb) {
  if (!texture) return {1, 1, 1};
  auto size = texture_size(texture);
  auto s = fmod(uv.x, 1) * size.x, t = fmod(uv.y, 1) * size.y;
  if (s < 0) s += size.x; if (t < 0) t += size.y;
  auto i=clamp((int)s,0,size.x-1), j=clamp((int)t,0,size.y-1);
  auto ii = (i + 1) % size.x, jj = (j + 1) % size.y;
  auto u = s - i, v = t - j;
  return lookup(texture,{ i,  j},no_srgb) * (1 - u) * (1 - v) +
         lookup(texture,{ i, jj},no_srgb) * (1 - u) * v +
         lookup(texture,{ii,  j},no_srgb) * u * (1 - v) +
         lookup(texture,{ii, jj},no_srgb) * u * v;
}
```

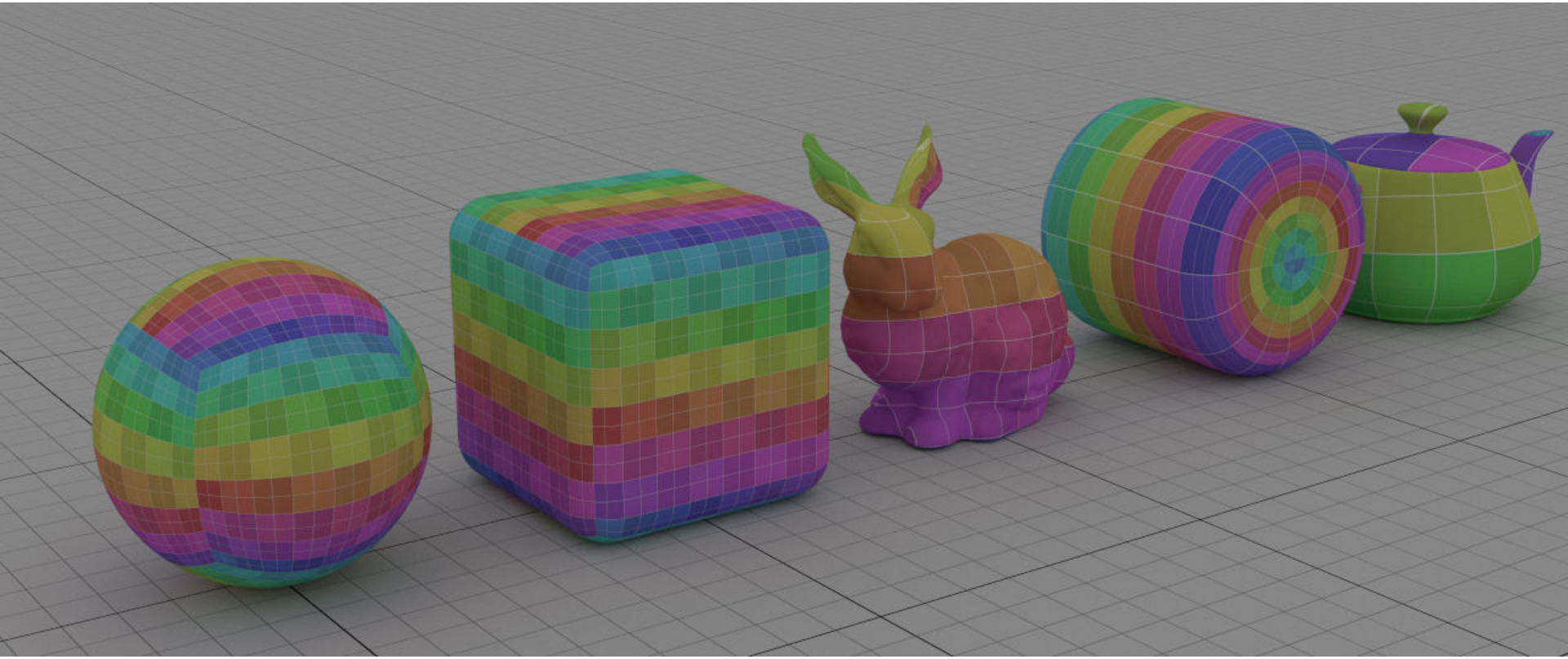

Lookup Texture Values

```
vec3f shade_texcoord(scene* scene, ray3f ray) {
 … return {fmod(eval_texcoord(shape, element, uv), 1), 0}; }
```



Lookup Texture Values

```
vec3f shade_indirect(scene* scene, ray3f ray) {
 … auto color = material->color *
  eval_texture(material->color_tex); …}
```



Lookup Environment Values

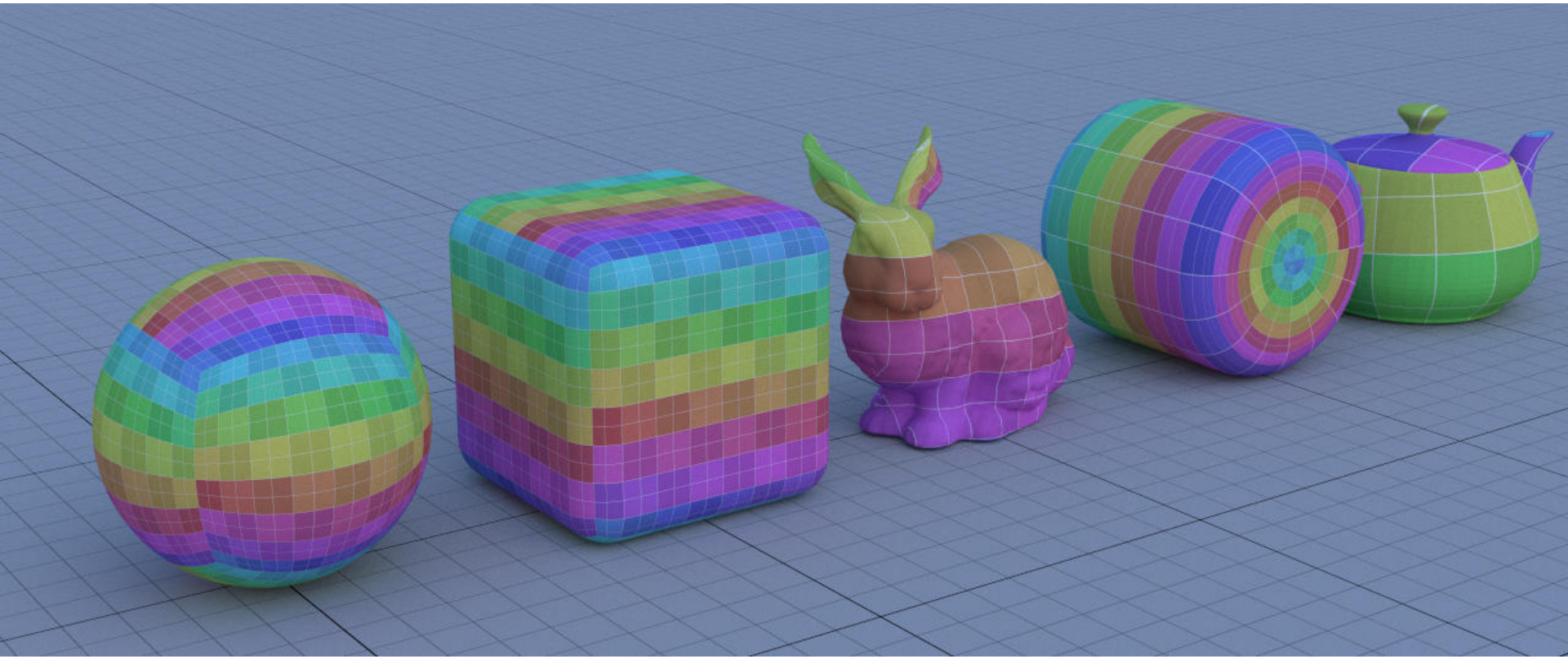
- Convert ray direction to latlong parametrization using spherical coordinates

$$u = \frac{\text{atan}(\mathbf{d}_z, \mathbf{d}_x)}{2\pi} \qquad v = \frac{\text{acos}(\mathbf{d}_y)}{\pi}$$

```
vec3f eval_environment(environment* environment, vec3f dir) {
  auto local_dir = transform_direction(
                       inverse(environment->frame), dir)
  auto texcoord = vec2f{
        atan2(local_dir.z, local_dir.x) / (2 * pi),
        acos(clamp(local_dir.y, -1, 1)) / pi};
  if (texcoord.x < 0) texcoord.x += 1;
  return environment->emission *
            eval_texture(environment->emission_tex, texcoord);
}
```

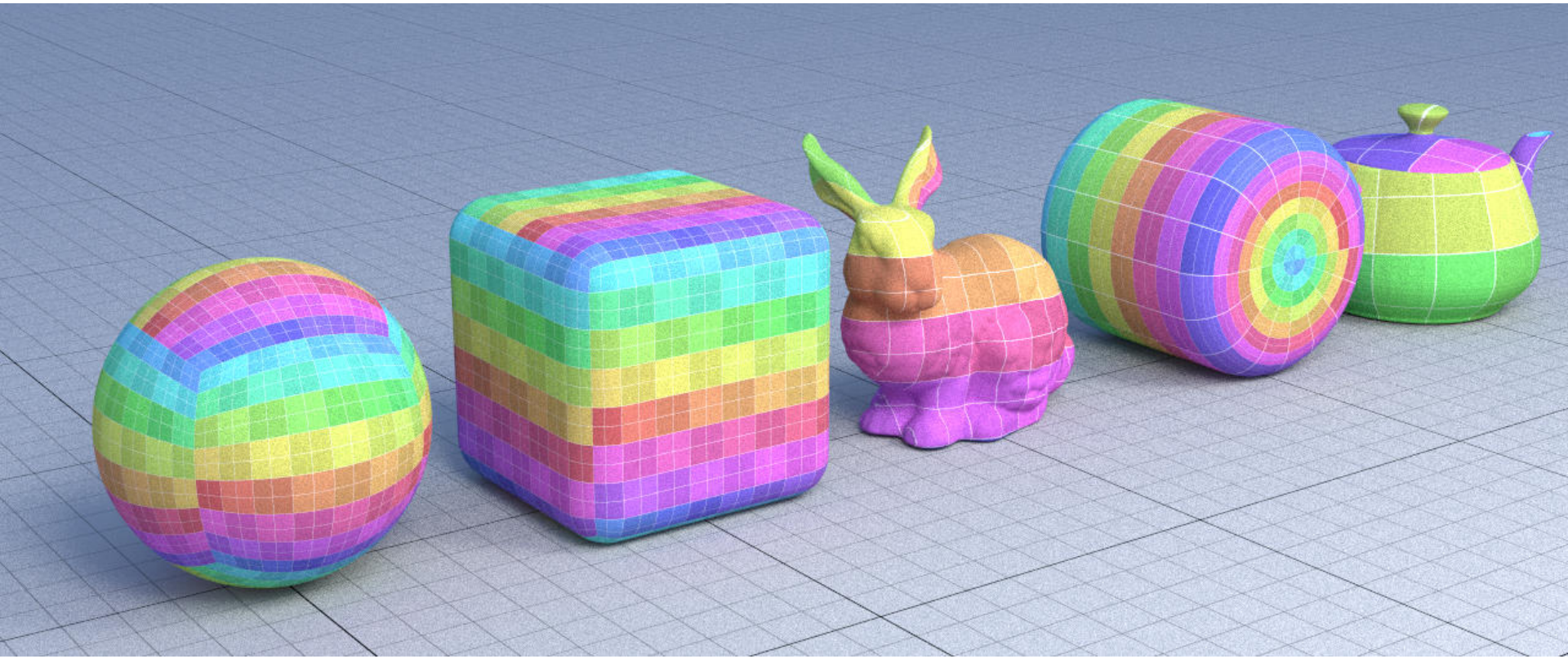

Lookup Environment Values

- Texture matte surfaces illuminated by a sky environment



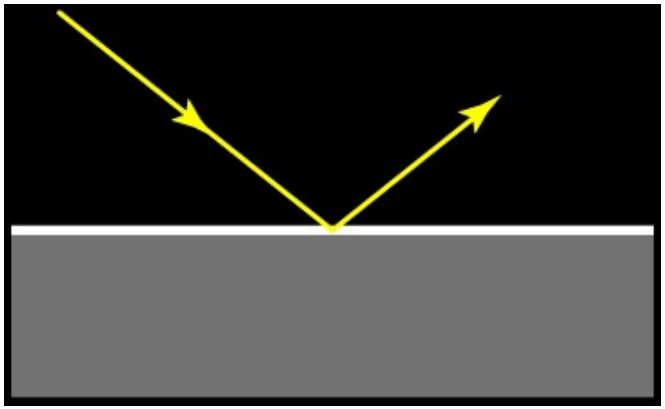
Lookup Environment Values

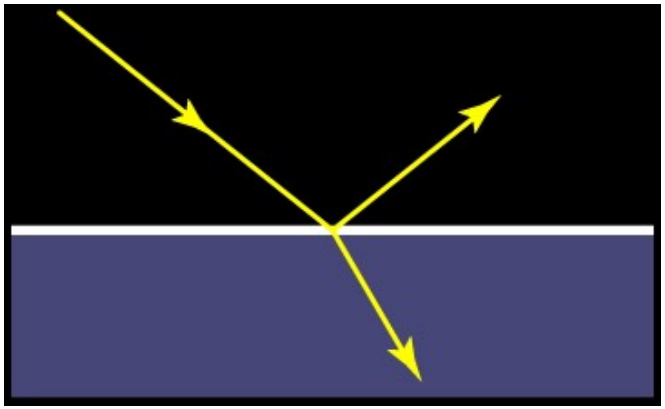
- Texture matte surfaces illuminated by a sky environment and area light



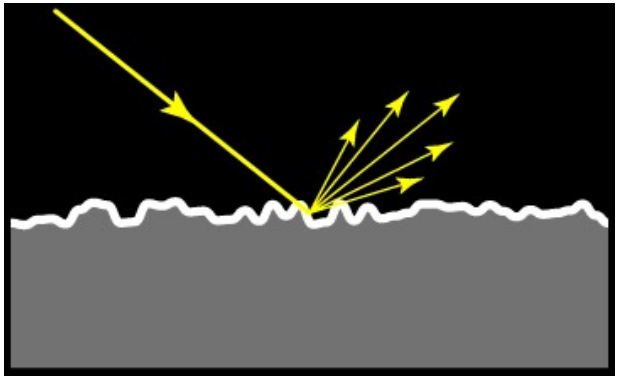
Materials

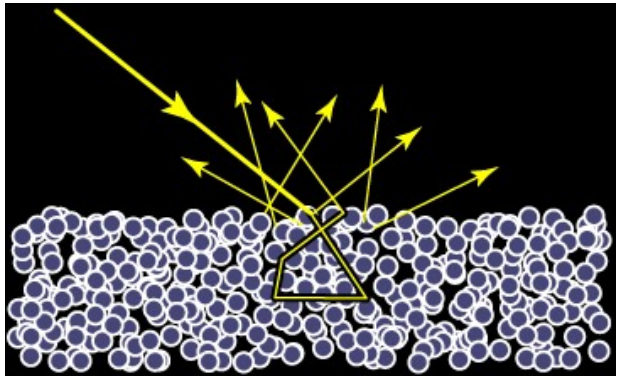
Polished surfaces





Rough surfaces



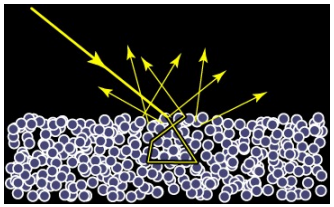


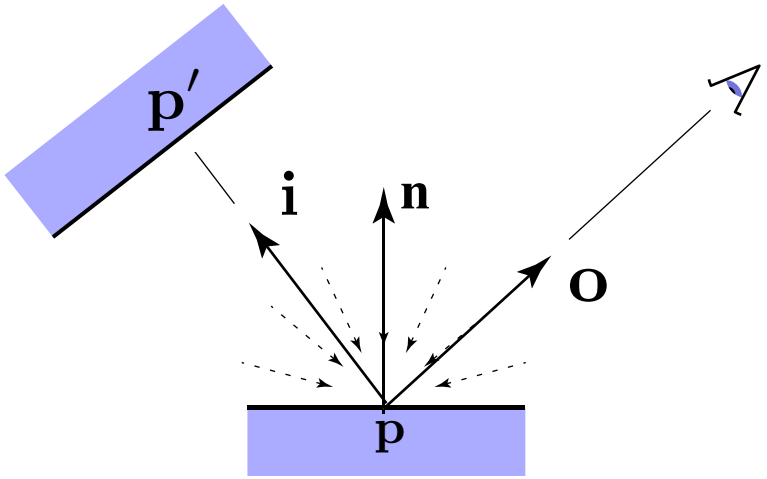
Handling multiple materials

```
vec3f shade_indirect(scene* scene, ray3f ray, int bounce) {
  auto isec = intersection3f{};
  if(!intersect_scene(scene, ray, &isec))
    return eval_environment(scene, ray.d);
  <compute position, normal, texcoord>
  <compute material values accounting for textures>
  <handle opacity>
  auto radiance = object->material->emission;
  if(bounce >= max_bounce) return radiance;
  if (transmission) { <handle polished dielectrics> }
  else if (metallic && !roughness) { <handle polished metals> }
  else if (metallic && roughness) { <handle rough metals> }
  else if (specular) { <handle rough plastic> }
  else { <handle diffuse> }
  return radiance;
}
```


Matte surfaces

- Matte surfaces scatter light in all directions
- They are approximated well by diffuse reflection
- We introduce a simplified notation to focus on material differences





$$L(\mathbf{p},\mathbf{o}) \approx \frac{2\pi}{n}\sum_{i}^{n}\frac{k_d}{\pi}L(\mathbf{p}'_i,-\mathbf{i}_i)|\mathbf{n}\cdot\mathbf{i}_i|$$

$$\mathbf{p}'_i = \text{intersect}(ray(\mathbf{p},\mathbf{i}_i))$$

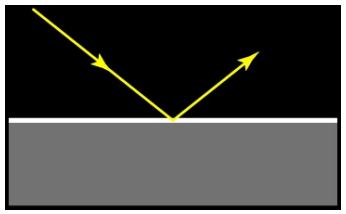
Matte surfaces

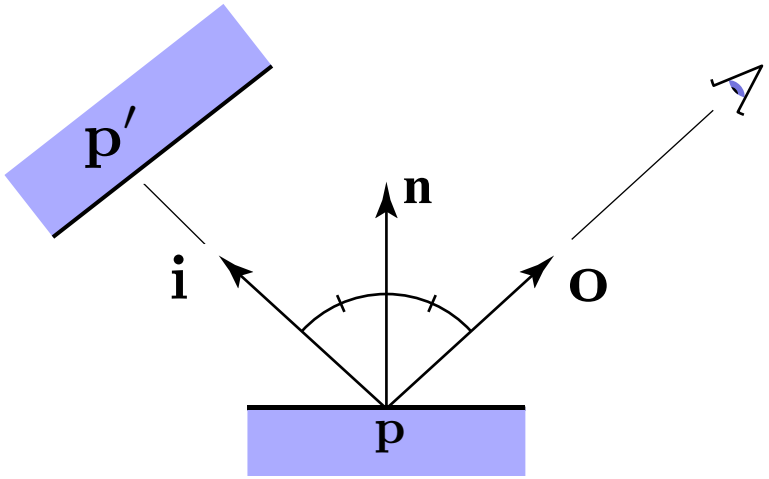
- Matte surfaces scatter light in all directions
- They are approximated well by diffuse reflection
- We introduce a simplified notation to focus on material differences

```
// same as before
<handle diffuse> {
  auto incoming = sample_hemisphere(normal, rand2f(rng));
  radiance += (2 * pi) * color / pi *
    shade_indirect(scene, ray3f{position, incoming},
        bounce+1) * dot(normal, incoming);
}
```


Polished mirrors

- Mirrors scatter light in exactly one direction
- They are approximated well by tracing a ray in the reflected direction





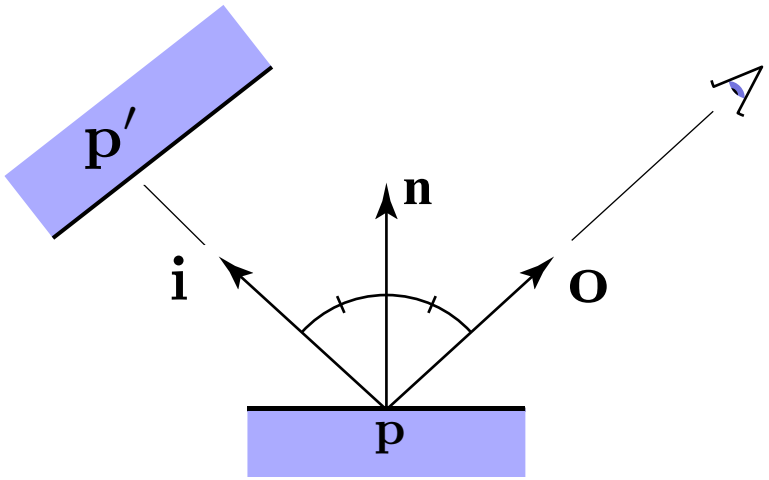
$$L(\mathbf{p}, \mathbf{o}) \approx k_s L(\mathbf{p}', -\mathbf{i})$$

$$\mathbf{p}' = \text{intersect}(ray(\mathbf{p}, \mathbf{i}))$$

$$\mathbf{i} = -\mathbf{o} + 2(\mathbf{o} \cdot \mathbf{n})\mathbf{n}$$

Polished metals

- Polished metals scatter light like mirrors
- The color of the surface though changes from the surface color at normal incidence to the color at grazing angle following Fresnel laws
- Here we show Schlick's approximation of the Fresnel term



$$L(\mathbf{p},\mathbf{o}) \approx F(k_s,\mathbf{n},\mathbf{o})L(\mathbf{p}',-\mathbf{i})$$

$$\mathbf{p}' = \text{intersect}(ray(\mathbf{p},\mathbf{i}))$$

$$\mathbf{i} = -\mathbf{o} + 2(\mathbf{o}\cdot\mathbf{n})\mathbf{n}$$

$$F(k_s,\mathbf{n},\mathbf{o}) = k_s + (1-k_s)(1-\mathbf{n}\cdot\mathbf{o})^5$$

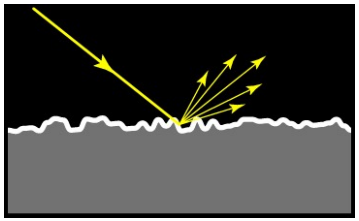
Polished metals

- Polished metals scatter light like mirrors
- The color of the surface though changes from the surface color at normal incidence to the color at grazing angle following Fresnel laws
- Here we show Schlick's approximation of the Fresnel term

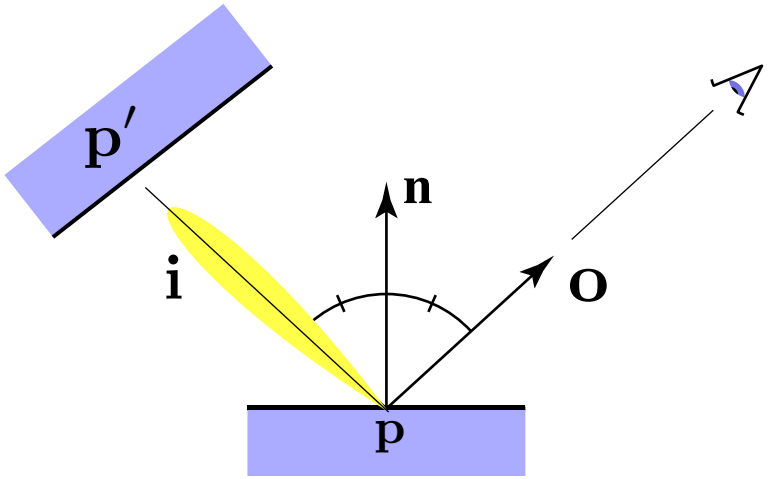
```
vec3f fresnel_schlick(vec3f ks, vec3f normal, vec3f outgoing) {
  return ks + (1 - ks)*pow(1 - dot(normal, outgoing), 5);
}

<handle polished metals> {
  auto incoming = reflect(outgoing, normal);
  radiance += fresnel_schlick(color, normal, outgoing)*
    shade_indirect(scene, ray3f{position, incoming},
        bounce+1);
}
```


Rough metals

- Rough metals scatter light around the reflected direction
- Their scattering is modeled by considered tiny micro-facets, each of which scatters light like a polished mirror



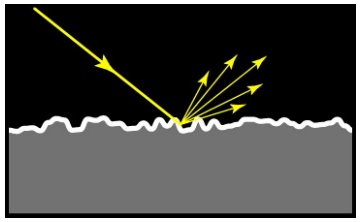
$$L(\mathbf{p},\mathbf{o}) \approx \frac{2\pi}{n}\sum_{i}^{n}\frac{FDG}{\pi|\mathbf{n}\cdot\mathbf{o}||\mathbf{n}\cdot\mathbf{i}|}L(\mathbf{p}'_i,-\mathbf{i}_i)|\mathbf{n}\cdot\mathbf{i}_i|$$



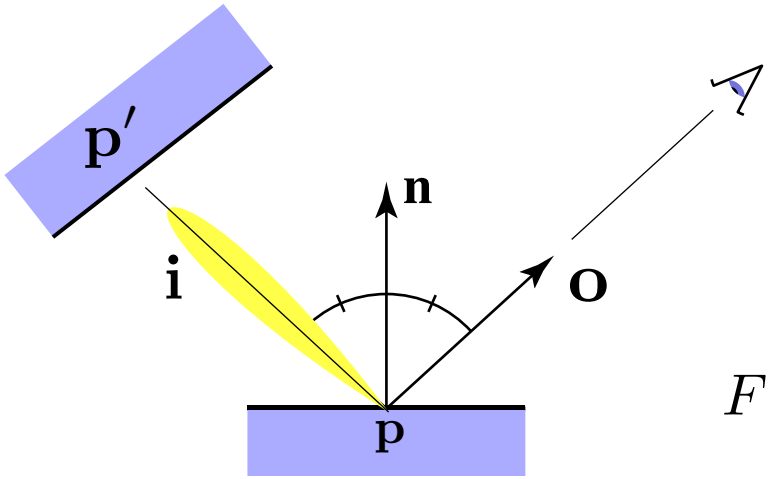
F: Fresnel term
D: microfacet distribution
G: microfacet shadowing

Rough metals

- The vector between the incoming and outgoing direction can be considered as an "effective" normal for the surface
- The Fresnel term is evaluated with respect to that



$$L(\mathbf{p},\mathbf{o}) \approx \frac{2\pi}{n}\sum_{i}^{n}\frac{FDG}{\pi|\mathbf{n}\cdot\mathbf{o}||\mathbf{n}\cdot\mathbf{i}|}L(\mathbf{p}'_i,-\mathbf{i}_i)|\mathbf{n}\cdot\mathbf{i}_i|$$

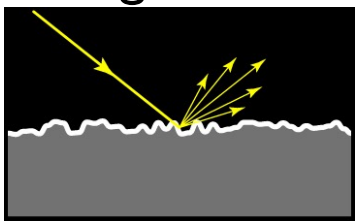


$$\mathbf{h} = \frac{\mathbf{o}+\mathbf{i}}{|\mathbf{o}+\mathbf{i}|}$$

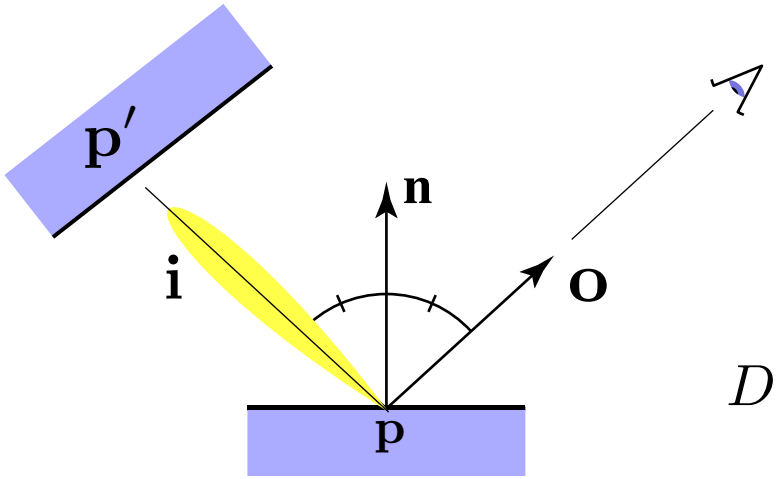
$$F(k_s,\mathbf{h},\mathbf{o}) = k_s + (1-k_s)(1-\mathbf{h}\cdot\mathbf{o})^5$$

Rough metals

- The microfacet distribution captures the ratio of microfacet that is oriented in a particular direction
- It depends on the normal and halfway directions and the surface roughness, that measure how wide the distribution is



$$L(\mathbf{p},\mathbf{o}) \approx \frac{2\pi}{n}\sum_{i}^{n}\frac{FDG}{\pi|\mathbf{n}\cdot\mathbf{o}||\mathbf{n}\cdot\mathbf{i}|}L(\mathbf{p}'_i,-\mathbf{i}_i)|\mathbf{n}\cdot\mathbf{i}_i|$$

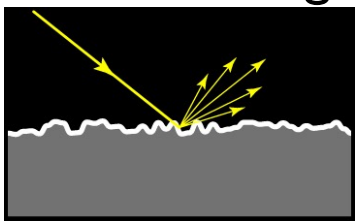


$$\mathbf{h} = \frac{\mathbf{o}+\mathbf{i}}{|\mathbf{o}+\mathbf{i}|}$$

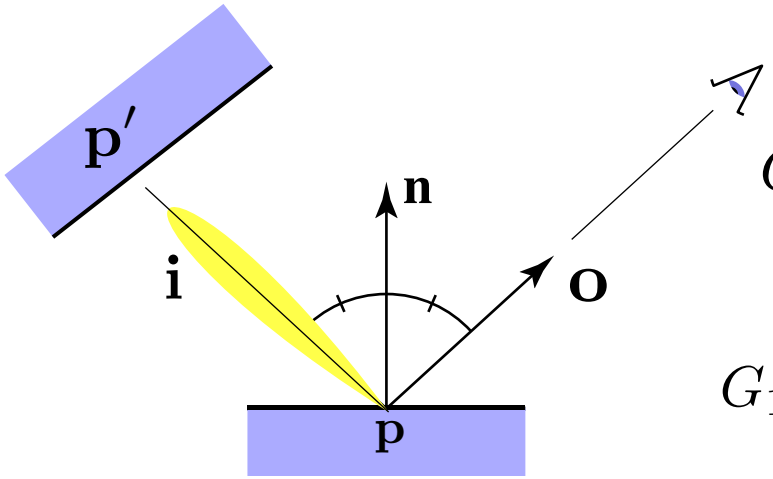
$$D(\mathbf{n},\mathbf{h},\alpha) = \frac{\alpha^2}{\pi((\mathbf{n}\cdot\mathbf{h})^2(\alpha^2-1)+1)^2}$$

Rough metals

- The shadowing term captures the ratio of microfacet that is visible from a particular incoming and outgoing angles
- It depends on the normal, incoming and outgoing directions and the surface roughness



$$L(\mathbf{p},\mathbf{o}) \approx \frac{2\pi}{n}\sum_{i}^{n}\frac{FDG}{\pi|\mathbf{n}\cdot\mathbf{o}||\mathbf{n}\cdot\mathbf{i}|}L(\mathbf{p}'_i,-\mathbf{i}_i)|\mathbf{n}\cdot\mathbf{i}_i|$$



$$G(\mathbf{n},\mathbf{o},\mathbf{i},\alpha) = G_1(\mathbf{n},\mathbf{o},\alpha)G_1(\mathbf{n},\mathbf{i},\alpha)$$

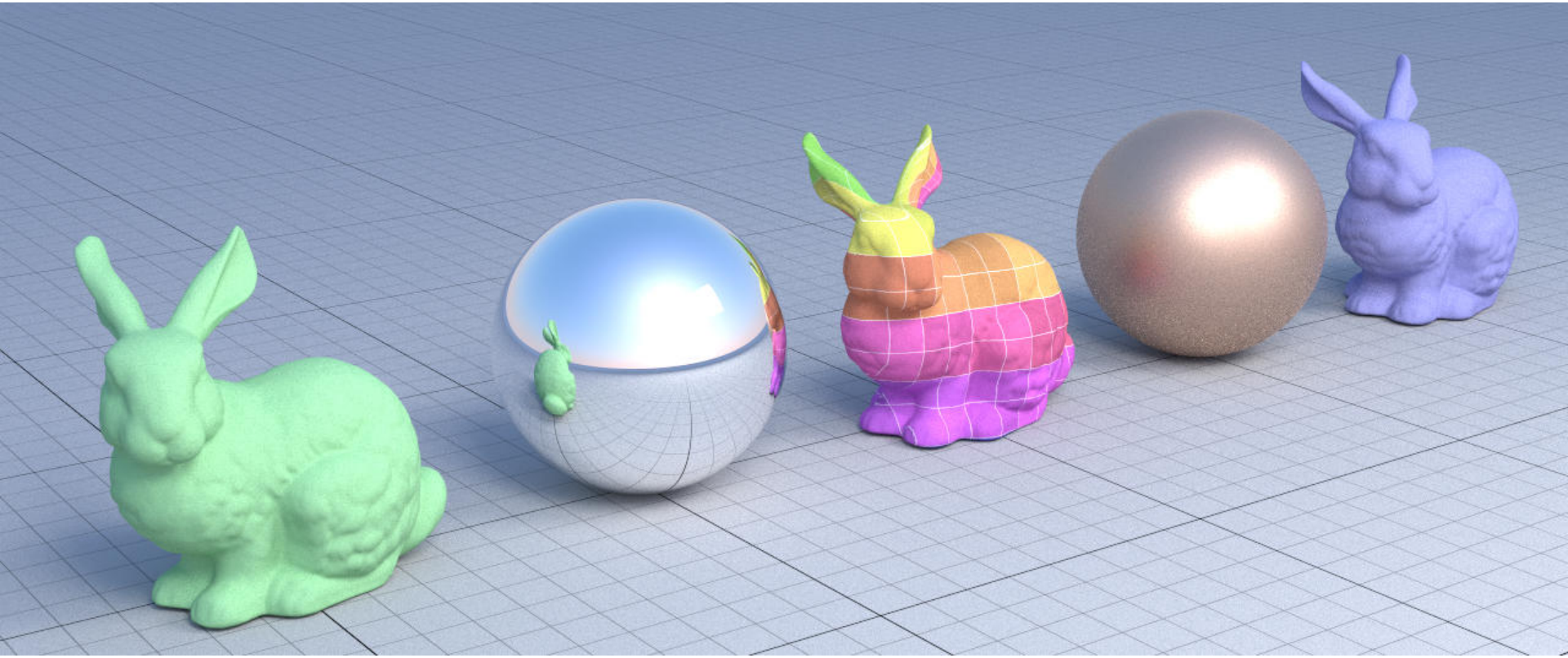
$$G_1(\mathbf{n},\mathbf{o},\alpha) = \frac{2\ \mathbf{n}\cdot\mathbf{o}}{\mathbf{n}\cdot\mathbf{o}+\sqrt{\alpha^2+(1-\alpha^2)(\mathbf{n}\cdot\mathbf{o})^2}}$$

Rough metals

- Rough metals scatter light around the reflected direction
- Their scattering is modeled by considered tiny micro-facets, each of which scatters light like a polished mirror

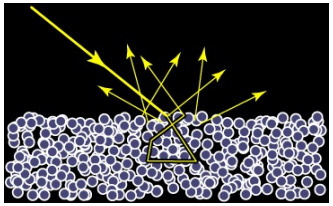
```
float microfacet_distribution(…) { … }
float microfacet_shadowing(…) { … }

<handle rough metals> {
  auto incoming = sample_hemisphere(normal, rand2f(rng));
  auto halfway = normalize(outgoing + incoming);
  radiance += (2 * pi) *
    fresnel_schlick(color, halfway, outgoing) *
    microfacet_distribution(roughness, normal, halfway) *
    microfacet_shadowing(roughness,normal,outgoing,incoming) /
    (4 * dot(normal, outgoing) * dot(normal, incoming)) *
    shade_indirect(scene, ray3f{position, incoming},
        bounce+1) * dot(normal, incoming);
}
```

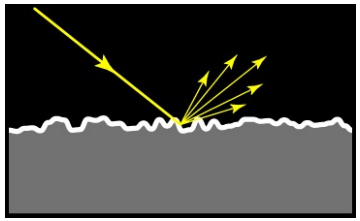
Polished and rough metals



Rough plastics

- Many surfaces can be modeled as a matte surface coated with a thin dielectric layer; an example of this are plastic surfaces
- They are approximated as sum of a diffuse and a specular contribution
- The specular layer reflects little light; 0.04 is a good number for it
- The diffuse component is weighted by one minus the contribution of the specular one to conserve evergy





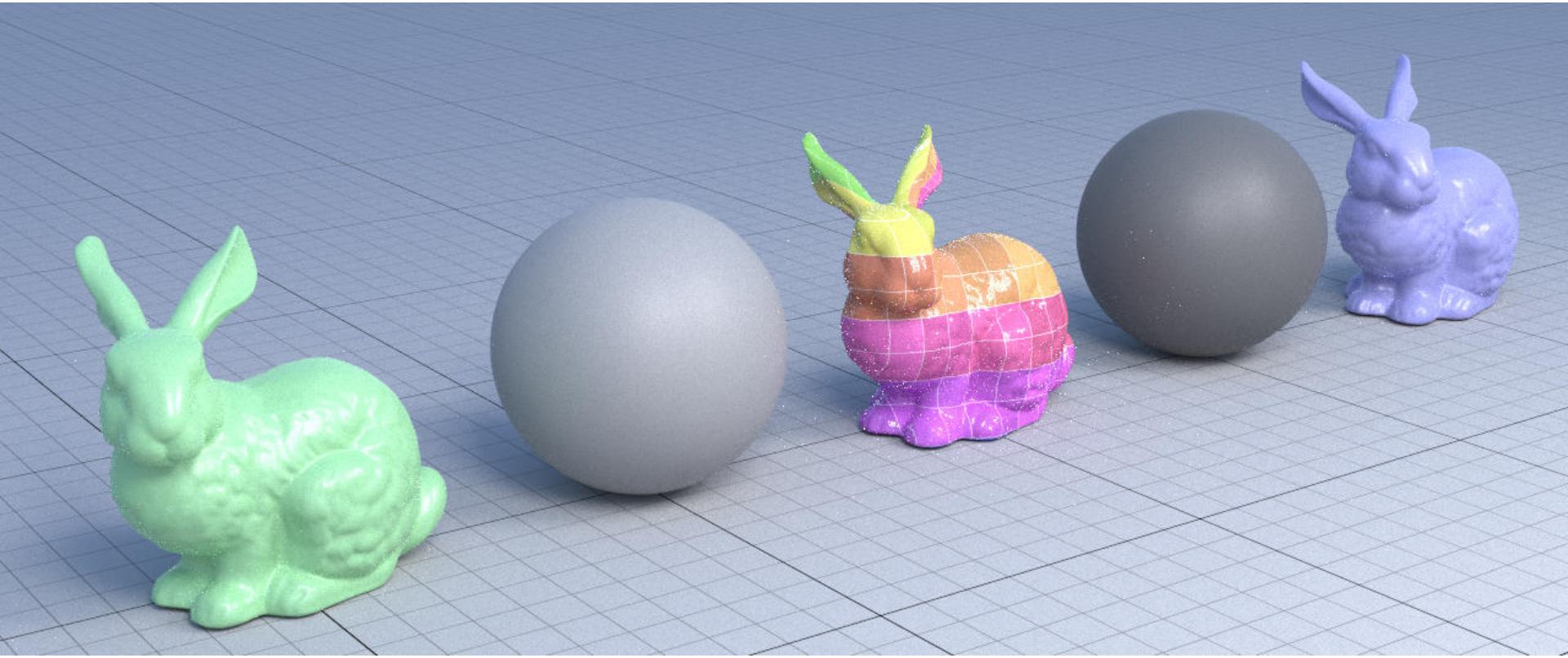
$$L(\mathbf{p},\mathbf{o}) \approx \frac{2\pi}{n}\sum_{i}^{n}\left[\frac{k_d(1-F(k_s,\mathbf{n},\mathbf{o}))}{\pi}+\frac{FDG}{\pi|\mathbf{n}\cdot\mathbf{o}||\mathbf{n}\cdot\mathbf{i}|}\right]L(\mathbf{p}'_i,-\mathbf{i}_i)|\mathbf{n}\cdot\mathbf{i}_i|$$

Rough plastics

- They are approximated as sum of a diffuse and a specular contribution

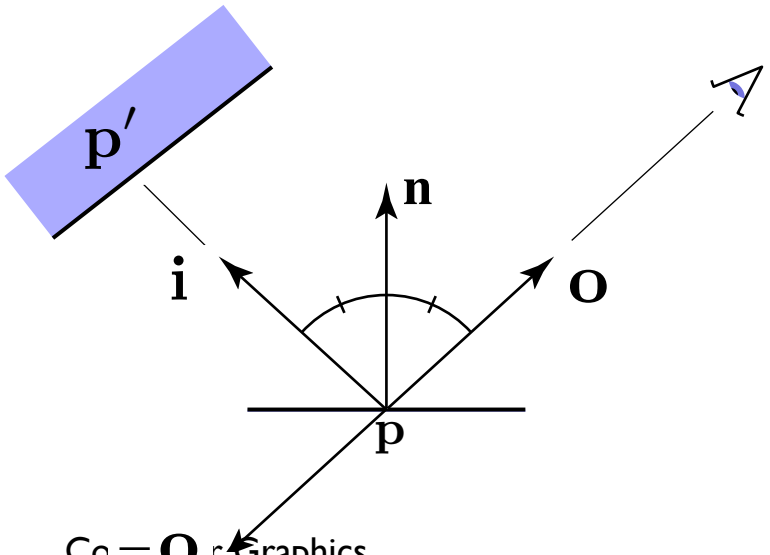
```
<handle rough plastic> {
  auto incoming = sample_hemisphere(normal, rand2f(rng));
  auto halfway = normalize(outgoing + incoming);
  radiance += (2 * pi) * (
    color / pi * (1 - fresnel_schlick(0.04,halfway,outgoing)) +
    fresnel_schlick(0.04, halfway, outgoing) *
    microfacet_distribution(roughness, normal, halfway) *
    microfacet_shadowing(roughness,normal,outgoing,incoming) /
    (4 * dot(normal, outgoing) * dot(normal, incoming))) *
    shade_indirect(scene, ray3f{position, incoming},
        bounce+1) * dot(normal, incoming);
}
```

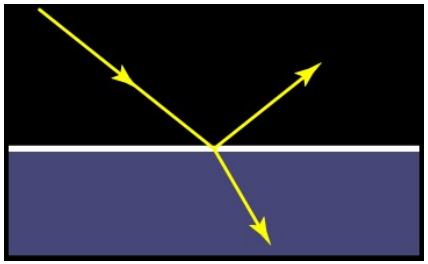

Rough plastics



Polished dielectrics

- Polished dielectrics scatter light both reflecting it and transmitting it
- We can model the reflection as polished metals but with a very small reflection coefficient; *ks* around 0.04
- The transmitted light is refracted into a material
- For thin objects, we can cheaply approximate this by letting the ray continue and weighting it by one minus the fresnel term





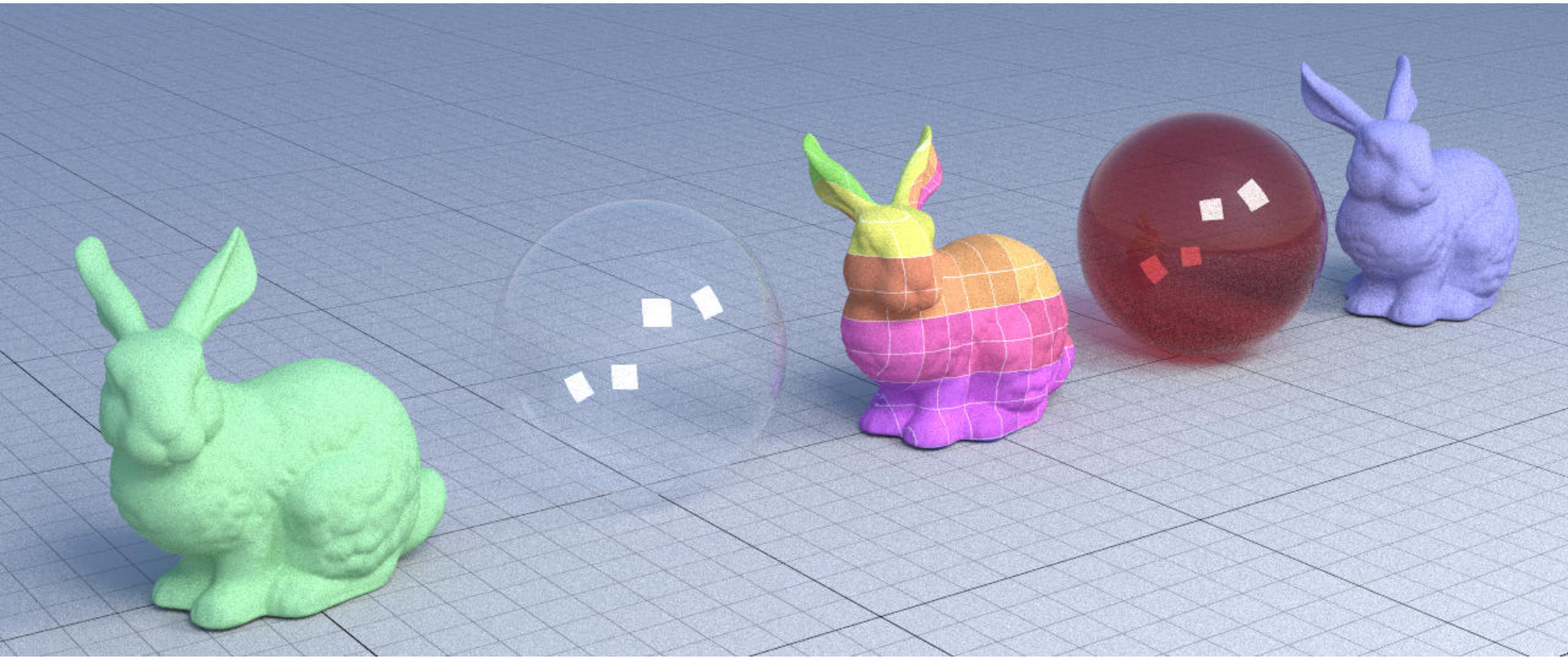
$$L(\mathbf{p},\mathbf{o}) \approx F(k_s,\mathbf{n},\mathbf{o})L(\mathbf{p}',-\mathbf{i}) + (1-F(k_s,\mathbf{n},\mathbf{o}))L(\mathbf{p}',-\mathbf{o})$$

Polished dielectrics

- Polished dielectrics scatter light both reflecting it and transmitting it
- We want to have only one ray continue, so we randomly pick which direction to go based on the fresnel term
- Due to this random picking, we do not need to reweighs for the Fresnel term

```
<handle polished dielectrics> {
  if(rand1f(rng) < fresnel_schlick(0.04, normal, outgoing)) {
    auto incoming = reflect(outgoing, normal);
    radiance += shade_indirect(scene, ray3f{position, incoming},
          bounce+1);
  } else {
    auto incoming = -outgoing;
    radiance += color *
      shade_indirect(scene, ray3f{position, incoming},
          bounce+1);
  }
}
```

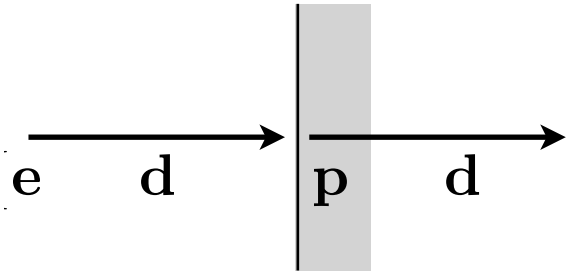
Polished dielectrics



Opacity

- As we saw for images, shapes can have an opacity stored in a texture
- This is helpful to model surfaces with holes without using triangles
- We will approximate this by continuing the ray is a random number is below the surface opacity

$$L_t = k_o \ \text{raytrace}(\{\mathbf{e}, \mathbf{d}\}) + (1 - k_o) \ \text{raytrace}(\{\mathbf{p}, \mathbf{d}\})$$

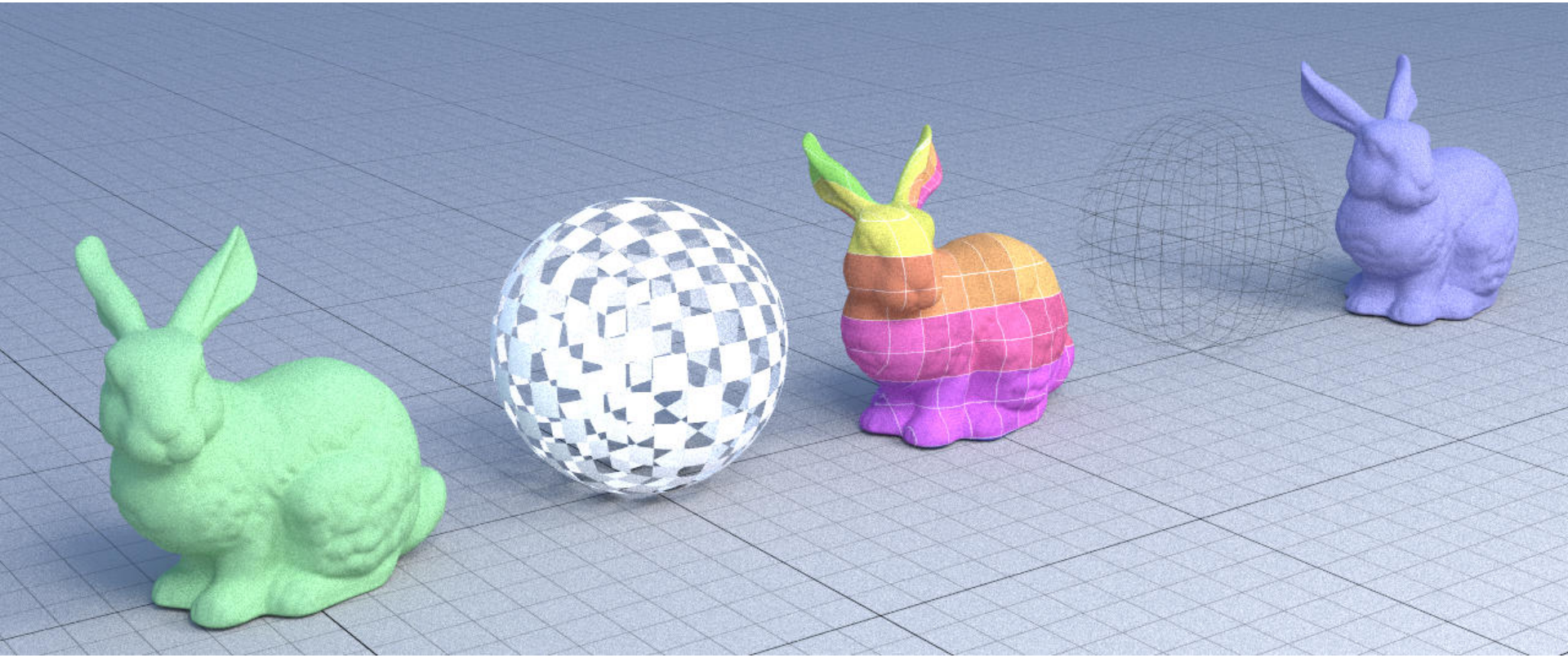


Opacity

- As we saw for images, shapes can have an opacity stored in a texture
- This is helpful to model surfaces with holes without using triangles
- We will approximate this by continuing the ray is a random number is above the surface opacity

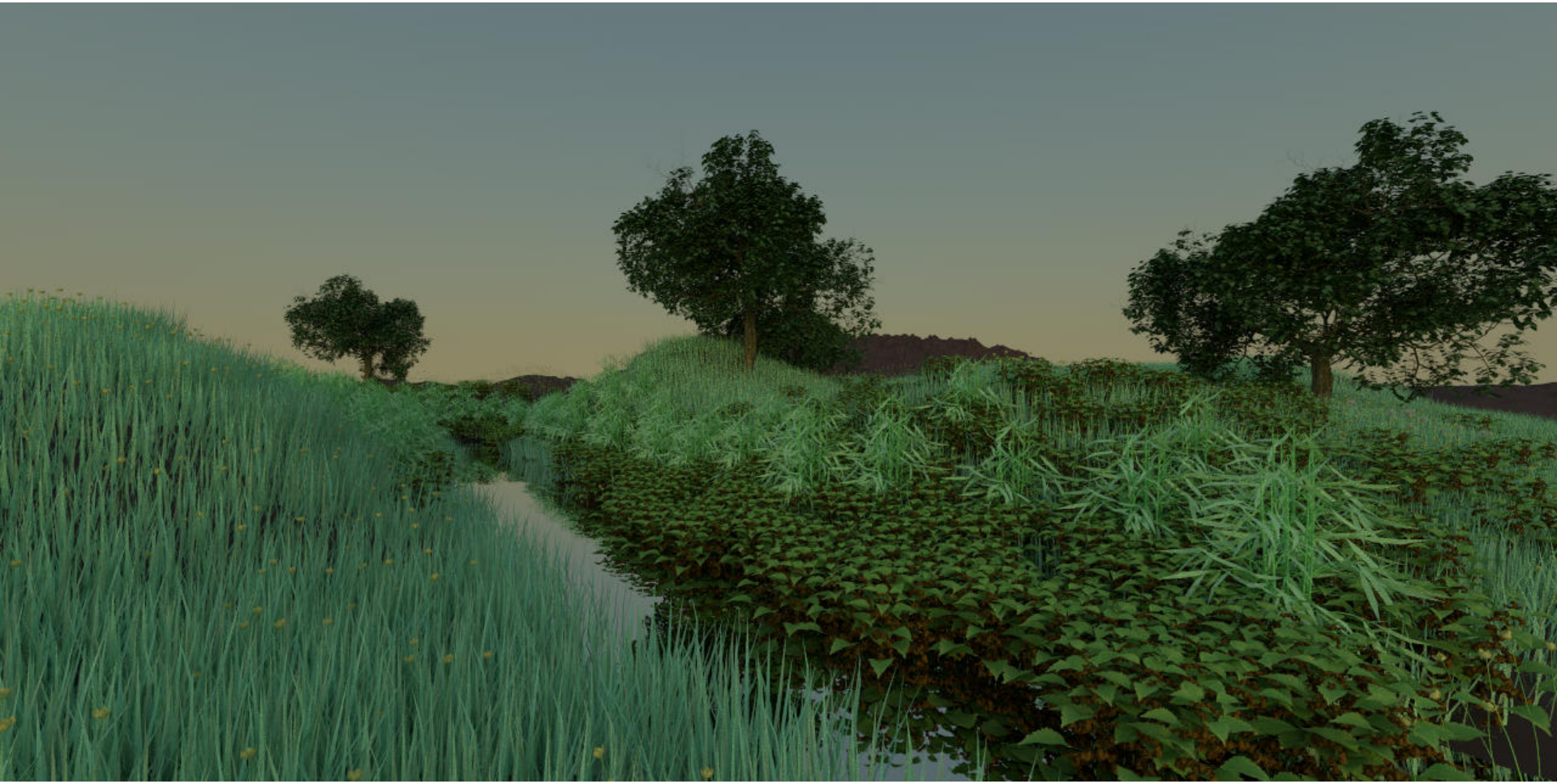
```
<handle opacity> {
  if(rand1f(rng) > opacity)
    return shade_indirect(scene, ray3f{position, outgoing},
          bounce+1);
}
```


Transparency



Final Images

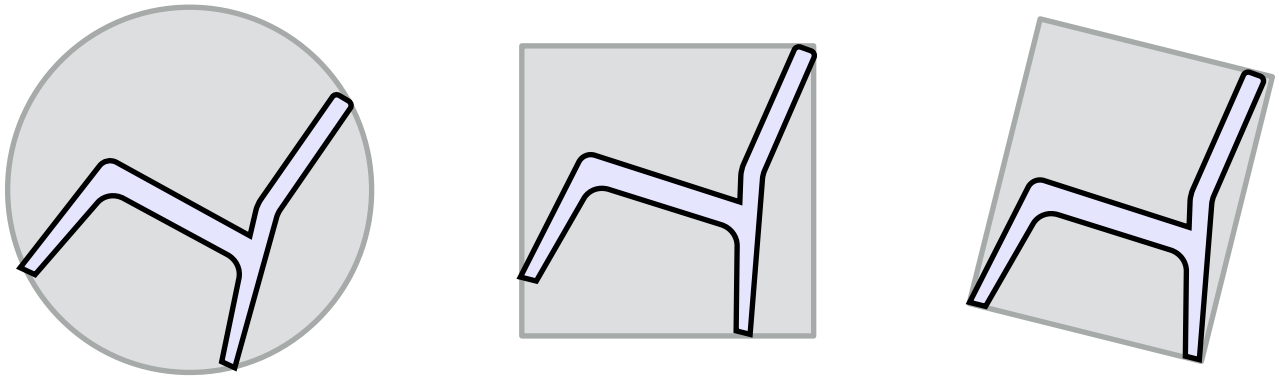
Final Images



Acceleration Intersections

Bounding volumes

- If the object is fully contained in a volume, then if the ray doesn't hit the volume, then it doesn't hit the object
- Quick way to avoid intersections: bound object with a simple volume, then test the bounding volume (BV) first, then test object if necessary

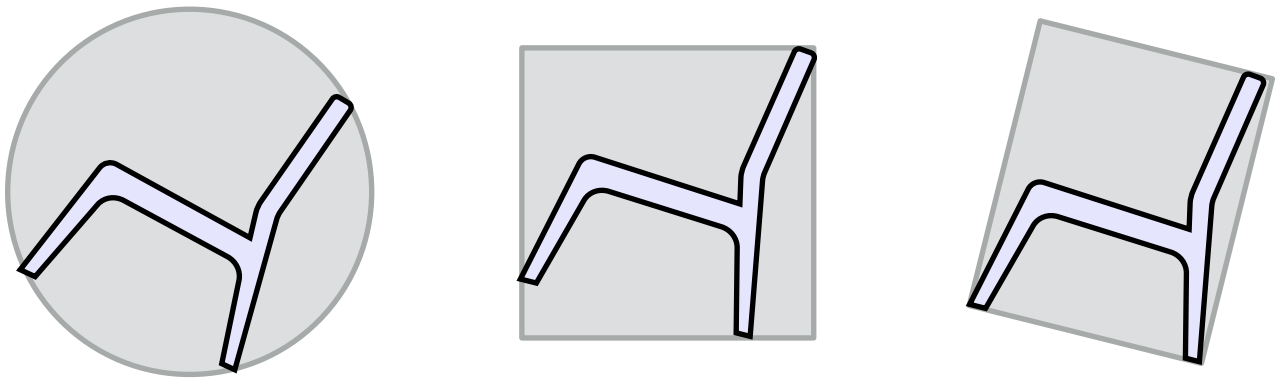


Bounding Volumes

- Cost: more for hits and near misses, less for far misses
- Cost of BV intersection test should be small
  - so use simple shapes (spheres, boxes, …)
- Cost of object intersect test should be large
  - so BV most useful for complex objects
- Tightness of fit should be good
  - loose fit leads to extra object intersections
  - tradeoff between tightness and BV intersection cost

Choice of bounding volumes

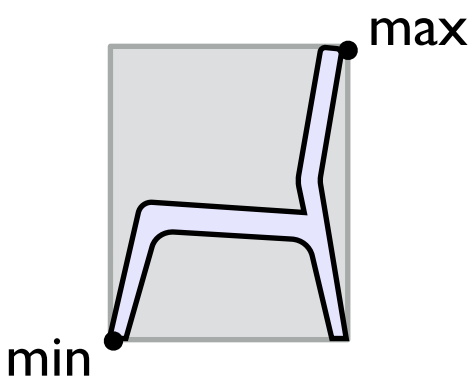
- Spheres: easy to intersect, not always so tight
  - hard to handle transformed meshes
- Axis-aligned bounding boxes (AABBs): easy to intersect, often tighter, esp. for axis-aligned models
- Oriented bounding boxes (OBBs): easy to intersect with some cost for transformation, tighter for arbitrary objects
  - not easy to compute a tight fir for meshes



Axis aligned bounding boxes

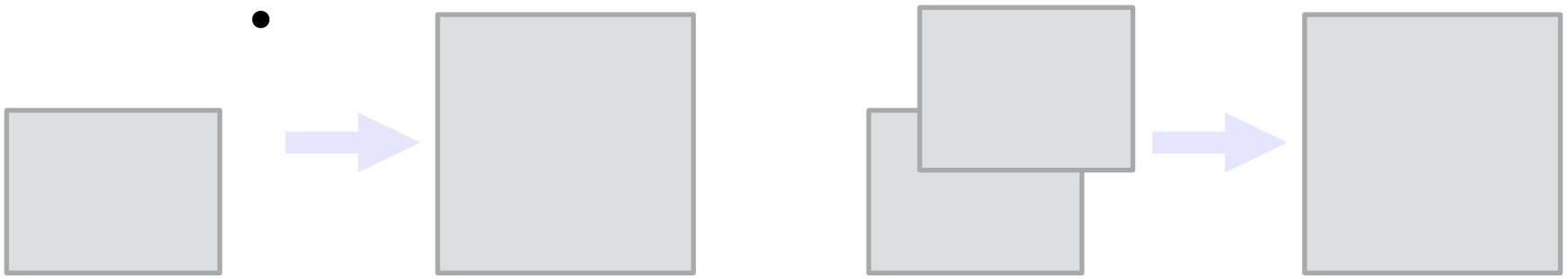
- Best tradeoff between simplicity and efficiency
- Store them by saving the minimum and maximum value in each coordinate

```
struct bbox3f {
  vec3f min, max;
};
```



Axis aligned bounding boxes

- Helpful operations: expand with either point or other box



```
bbox3f merge(bbox3f a, vec3f b) {
  return { min(a.min, b), max(a.max, b) };
}

bbox3f merge(bbox3f a, bbox3f b) {
  return { min(a.min, b.min), max(a.max, b.max) };
}
```

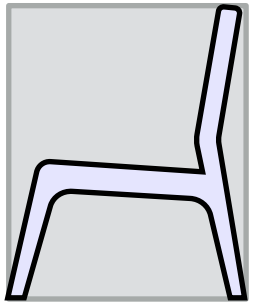

Construct AABBs

- For triangles just take the min and max of its vertices
- For parts of meshes: min and max of vertices

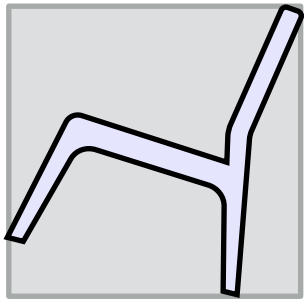
```
bbox3f triangle_bound(vec3f v0, vec3f v1, vec3f v2) {
  auto box = invalidb3f; // min==flt_max, max==flt_min
  box = expand_bbox(v0);
  box = expand_bbox(v1);
  box = expand_bbox(v2);
  return box;
};
```


Construct AABBs

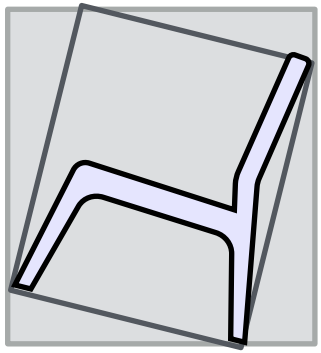
- AABBs for a surface instance
  - Transform all points for tight bound — but very inefficient
  - Most commonly take the AABB of the 8 corners of the AABB of the untransformed surface



untransformed



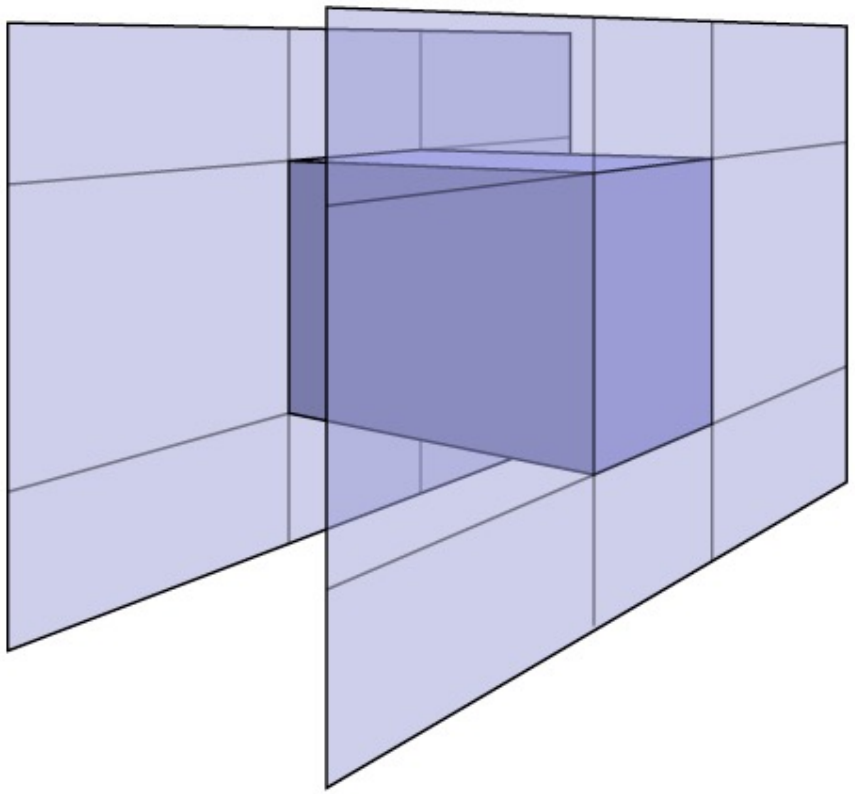
bounding transformed vertices



bounding transformed corners

Ray-box intersection

- Could intersect with 6 faces individually
- Better way: box is the intersection of 3 slabs

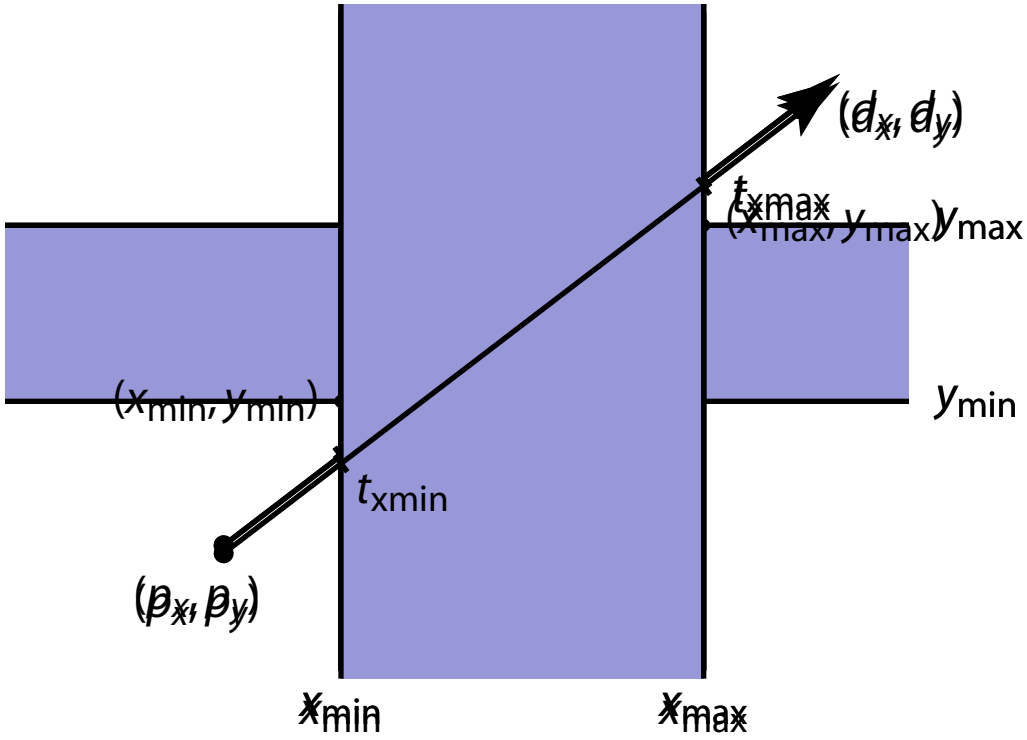


Ray-slab intersection

- Show in 2D since 3D is the same

$$p_x + t_{x\min}\, d_x = x_{\min}$$
$$t_{x\min} = (x_{\min} - p_x)/d_x$$

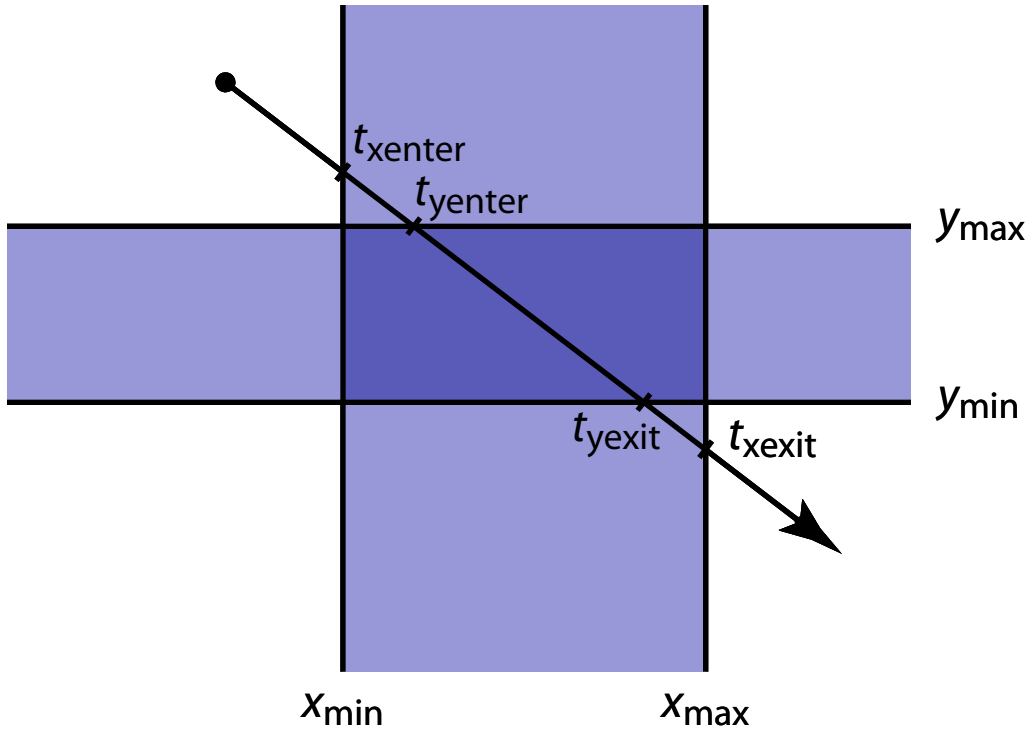
$$p_y + t_{y\min}\, d_y = y_{\min}$$
$$t_{y\min} = (y_{\min} - p_y)/d_y$$



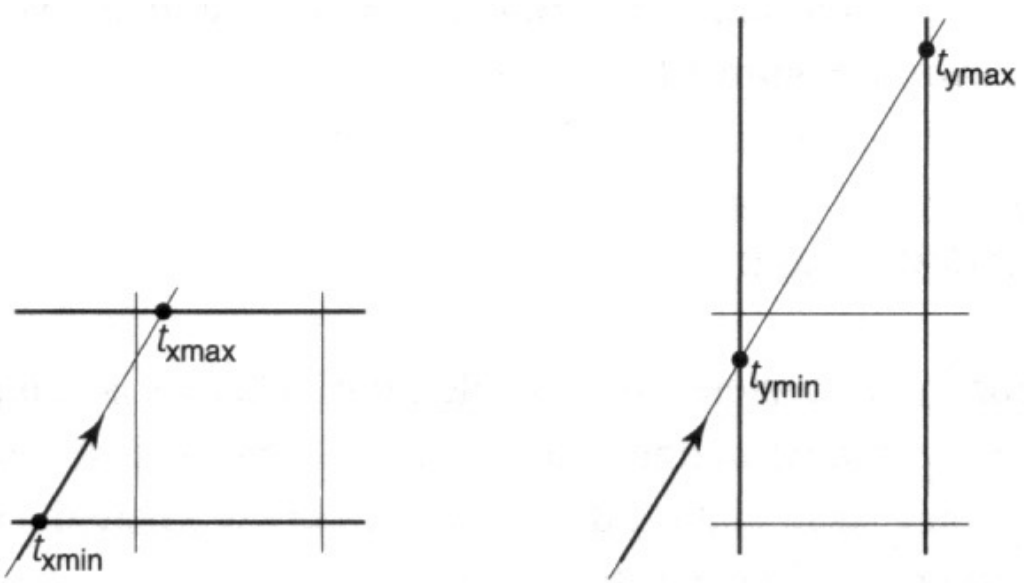
Ray-box intersection

- Each intersection is an interval
- Want last entry point and first exit point

$$t_{x\text{enter}} = \min(t_{x\text{min}}, t_{x\text{max}})$$
$$t_{x\text{exit}} = \max(t_{x\text{min}}, t_{x\text{max}})$$
$$t_{y\text{enter}} = \min(t_{y\text{min}}, t_{y\text{max}})$$
$$t_{y\text{exit}} = \max(t_{y\text{min}}, t_{y\text{max}})$$
$$t_{\text{enter}} = \max(t_{x\text{enter}}, t_{y\text{enter}})$$
$$t_{\text{exit}} = \min(t_{x\text{exit}}, t_{y\text{exit}})$$



Ray-box intersection



$t \in [\, t_{xmin}, t_{xmax} \,]$

$t \in [\, t_{ymin}, t_{ymax} \,]$

$t \in [\, t_{xmin}, t_{xmax} \,] \cap [\, t_{ymin}, t_{ymax} \,]$

Ray-box intersection

```
// implementation 1
bool intersect_bbox(ray3f ray, bbox3f bbox) {
  auto invd = vec3f{1/ray.d.x,1/ray.d.y,1/ray.d.z};
  auto t0 = (bbox.min - ray.o) * invd;
  auto t1 = (bbox.max - ray.o) * invd;
  if (invd.x < 0) std::swap(t0.x, t1.x);
  if (invd.y < 0) std::swap(t0.y, t1.y);
  if (invd.z < 0) std::swap(t0.z, t1.z);
  auto tmin = max(t0.z, max(t0.y, max(t0.x, ray.tmin)));
  auto tmax = min(t1.z, min(t1.y, min(t1.x, ray.tmax)));
  tmax *= 1.00000024f; // numerical precision fix
  return tmin <= tmax;
}
```

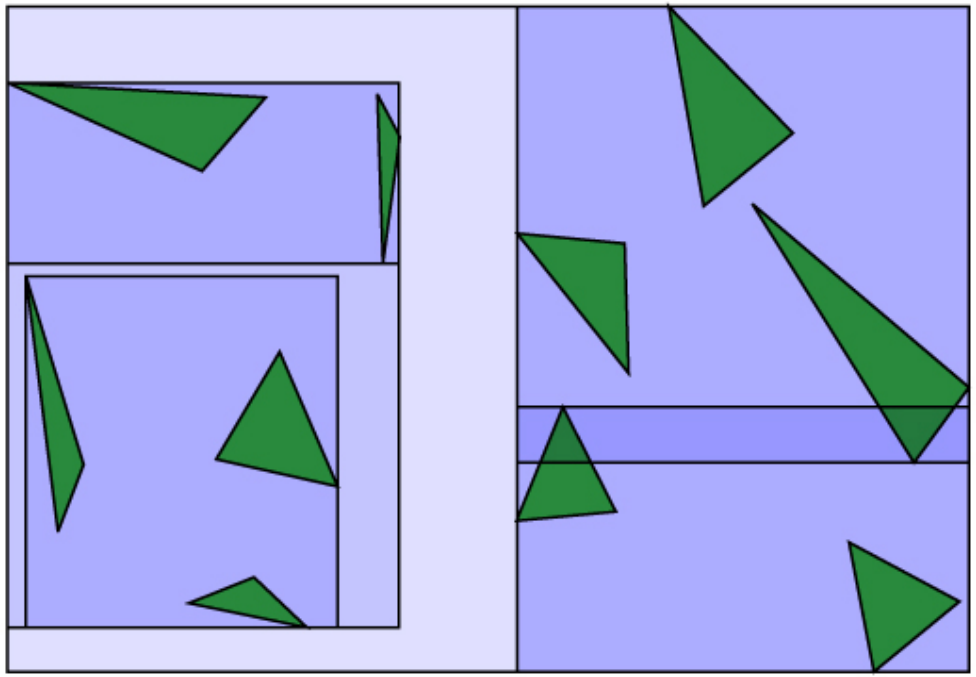

Ray-box intersection

```
// implementation 2
bool intersect_bbox(ray3f ray, vec3f rayd_inv, bbox3f bbox) {
  auto hit_min = (bbox.min - ray.o) * ray_dinv;
  auto hit_max = (bbox.max - ray.o) * ray_dinv;
  auto tmin    = min(hit_min, hit_max);
  auto tmax    = max(hit_min, hit_max);
  auto t0      = max(max(tmin), ray.tmin);
  auto t1      = min(min(tmax), ray.tmax);
  t1 *= 1.00000024f;
  return t0 <= t1;
}
```

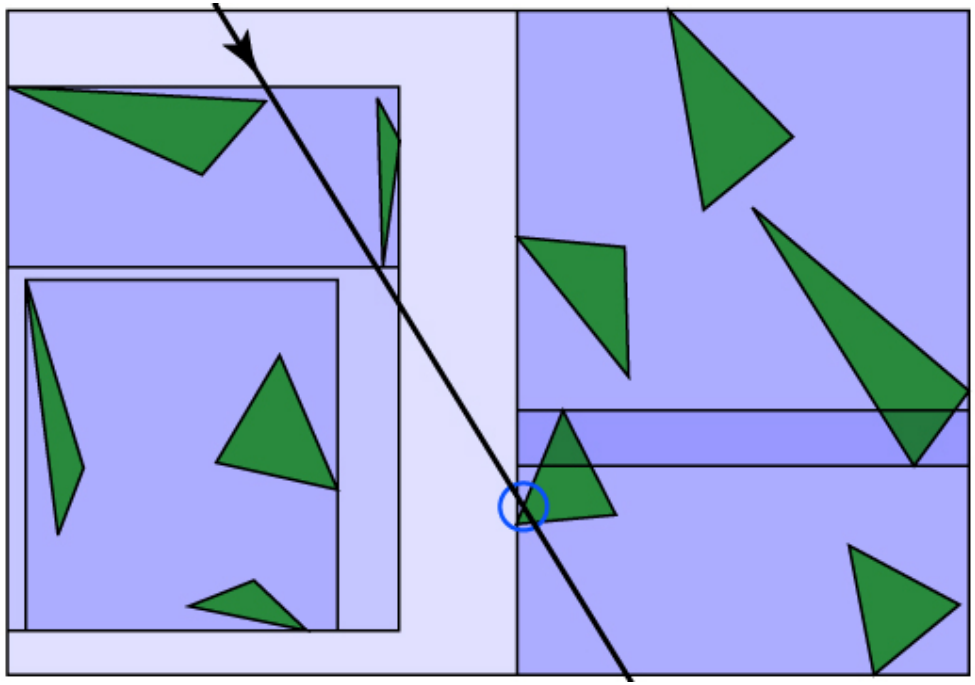

Bounding Volume Hierarchy

- Bounding box around a mesh will help
- Bounding box around groups of triangles will help
- Bounding box around groups of meshes will help
- Leads to the idea of using bounding volumes all the way from the whole scene down to groups of a few objects
- BVH: bounding volume hierarchy

BVH construction example



BVH ray-tracing example



Implementing BVHs

- For efficiency, avoid trees with pointers
- Use a flattened, or linear, BVH where all nodes are stored in one array and all primitives in another
- Internal nodes have indices to the node array
- Leaf nodes have indices to the leaves

```
struct bvh_node {
  bbox3f bbox;
  uint32 start;
  uint16 num;
  uint16 internal;
};
```

```
struct bvh_tree {
  vector<bvh_node> nodes;
  vector<int> primitives;
};
```



```
bool intersect_bvh(bvh_tree* bvh, shape* shape, ray3f ray,
                   intersection* isec) {
  int node_stack[64]; auto node_cur = 0;
  node_stack[node_cur++] = 0;
  auto hit = false;
  while (node_cur) {
    auto node = bvh->nodes[node_stack[--node_cur]];
    if (!intersect_bbox(ray, node.bbox)) continue;
    if (node.internal) {
       for (auto i = 0; i < node.count; i++)
         node_stack[node_cur++] = i + node.start;
    } else {
      for (auto i = 0; i < node.count; i++) {
        auto t = shape->triangles[bvh->primitives[i]];
        if (!intersect_triangle(ray, shape->positions[t.x],
            shape->positions[t.y], shape->positions[t.z],
            isec->distance, isec->uv)) continue;
        hit = true; ray.tmax = dist;
        isec->element = bvh->primitives[i];
      }
    }
  return hit;
}
```


Building a hierarchy

- Usually do it top-down
- Make box for whole scene, then split into two parts
- Recurse on parts, stopping when there are too few objects
- Split by sorting along longest axis
- Many heuristics
  - split in the middle of the parent box — split space, but not number of primitives
  - split into two equal number of components — split primitives, but may leads to overlapping boxes
  - surface area heuristic: split for best ray cost — hard to implement but leads to better intersection


```
// generic primitive
struct bvh_prim {
  bbox3f bbox;   // bounding box
  vec3f center;  // bounding box center
  int primitive; // primitive id
};

void build_bvh(bvh_tree* bvh, shape* shape, bool equal_num) {
  auto prims = std::vector<bvh_prim>();
  for (auto ei = 0; ei < shape->triangles.size(); ei++) {
    auto e = shape->triangles[ei];
    auto bbox = triangle_bound(shape->positions[e.x],
      shape->positions[e.y], shape->positions[e.z]);
    prims.push_back({bbox, (bbox.min + bbox.max) / 2, ei});
  }
  build_bvh(bvh, bound_prims, equal_num);
}
```



```
// recursively build a BVH by partitioning and sorting
// the prim array
void build_bvh(bvh_tree* bvh, vector<bvh_prim>& bound_prims,
    bool equal_num) {
  // prepare the tree and repallocate memory
  bvh->nodes.reserve(prims.size() * 2);
  bvh->nodes.push_back({});
  // recursively partition and sort the prim array
  make_node(bvh, 0, prims, 0, prims.size(), equal_num);
  bvh->nodes.shrink_to_fit();
  // copy the leaf nodes back
  bvh->leaf_prims.resize(prims.size());
  for (int i = 0; i < prims.size(); i++) {
    bvh->leaf_prims[i] = bound_prims[i].pid;
  }
  // done
  return bvh;
}
```



```
void make_node(bvh_tree* bvh, int node_id,
    vector<bound_prim>&  prims, int start, int end) {
    auto node = &bvh->nodes[node_id];            // grab node
    node->bbox = invalidb3f;                     // compute bbox
    for (auto i = start; i < end; i++)
        node->bbox = merge(node->bbox, prims[i].bbox);
    auto split = false; auto axis = 0; auto mid = (start+end)/2;
    if (end - start > 4)        // split prims and get mid point
        split = split_prims(prims,start,end,&axis,&mid);
    if(!split) {                                  // make a leaf
        node->internal = false;
        node->start = start;
        node->count = end-start;
    } else {                              // make an internal node
        node->internal = true;
        node->axis = axis;
        auto first = (int)bvh->nodes.size();
        node->start = first;
        node->count = 2;
        // allocate children and recurse
        bvh->nodes.push_back({}); bvh->nodes.push_back({});
        make_node(bvh, first, prims, start, mid);
        make_node(bvh, first + 1, prims, mid, end);
    }
}
```



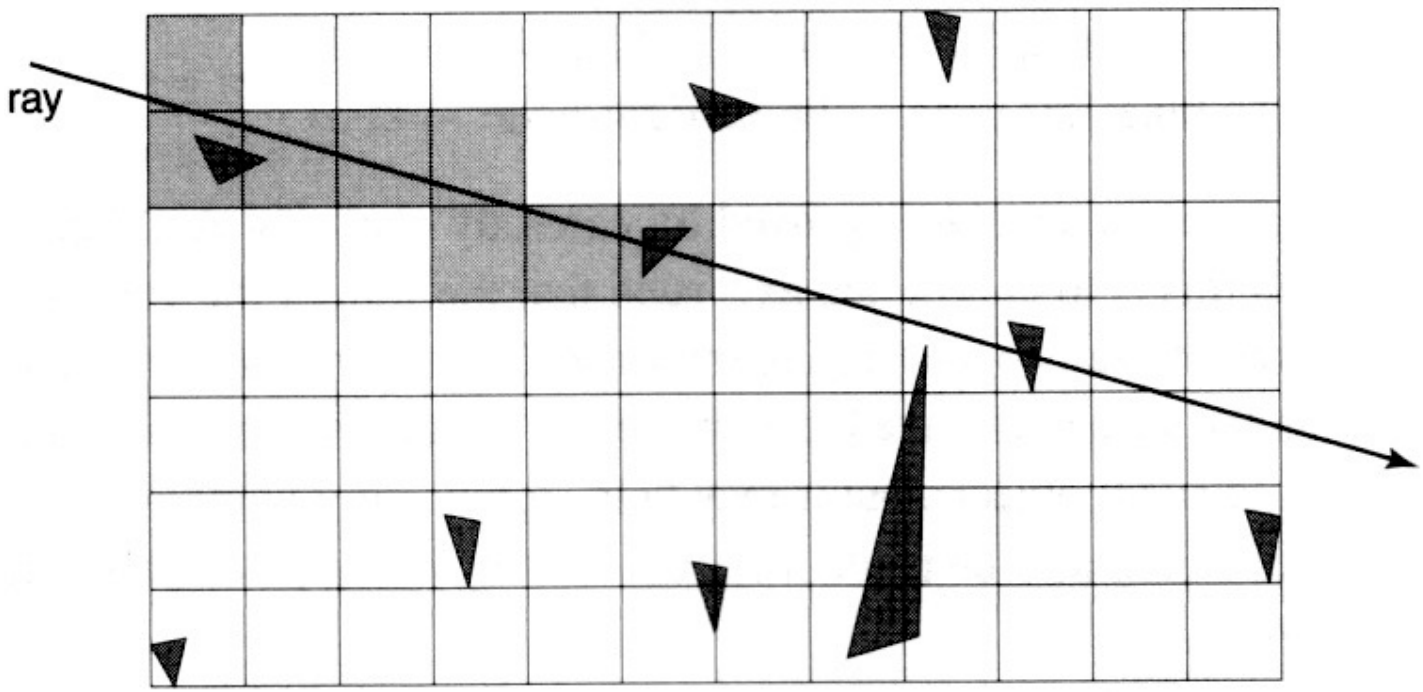
```
bool split_prims_equalnum(vector<bound_prim>& prims,
    int start, int end, int* axis, int* mid) {
  // pick axis by choosing the largest axis of the box
  // use centers to avoid issues with large primitives
  auto cbbox = invalidb3f;
  for (auto i = start; i < end; i++)
    cbbox = merge(cbbox, prims[i].center);
  auto size = cbbox.max - cbbox.min;
  if (size == vec3f{0, 0, 0}) return false;
  // pick axis and mid
  *axis = max_index(size); // pick largest axis
  *mid = (start + end) / 2; // split elements in half
  // arrange the array to have elements divided in two groups
  // w.r.t the middle element; smaller on the left, and larger
  // on the right; C++ std::nth_element does that
  std::nth_element(prims.begin() + start,
                   prims.begin() + *mid,
                   prims.begin() + end,
                   [axis](auto a, auto b) {
                     return a.center[*axis] < b.center[*axis];
                   });
  return true;
}
```



```
bool split_prims_equalsize(vector<bound_prim>& prims,
    int start, int end, int* axis, int* mid) {
  // pick axis by choosing the largest axis of the box
  // use centers to avoid issues with large primitives
  auto cbbox = invalidb3f;
  for (auto i = start; i < end; i++)
    cbbox = merge(cbbox, prim[i].center);
  auto csize = cbbox.max - cbbox.min;
  if (csize == vec3f{0, 0, 0}) return false;
  // pick axis and split center
  *axis = max_index(csize); // pick largest axis
  // split space in half by the half plane of the bbox
  auto half = (cbbox.min + cbbox.max) / 2;
  // partition the array such that all left elements are
  // less or equal to half and all right elements are greater;
  // compute the split point mid; C++ std::partition does that
  *mid = (int)(std::partition(prims.begin() + start,
        prims.begin() + end,
        [axis, half](auto a) {
            return a.center[*axis] < half[*axis];
        }) - prims.begin());
  return true;
}
```

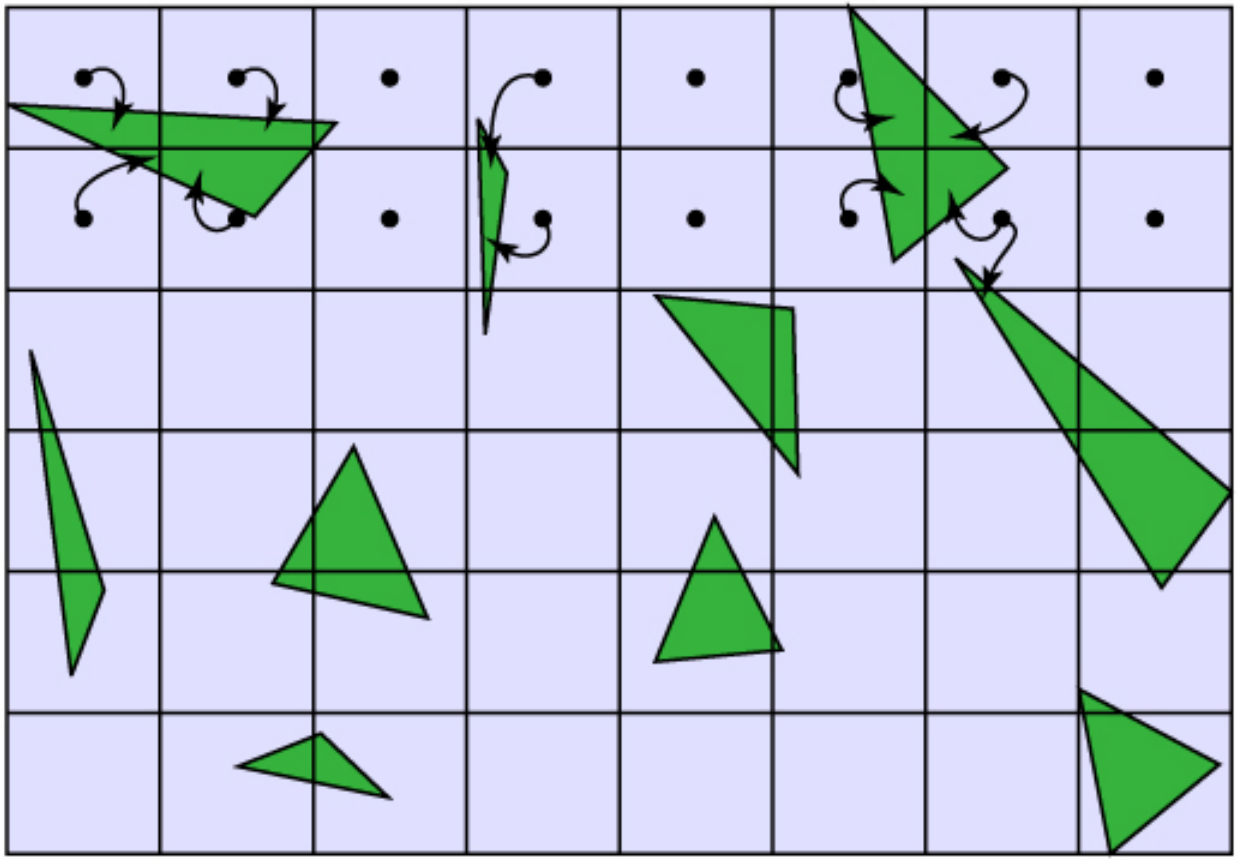

Regular space subdivision

- An entirely different approach: uniform grid of cells

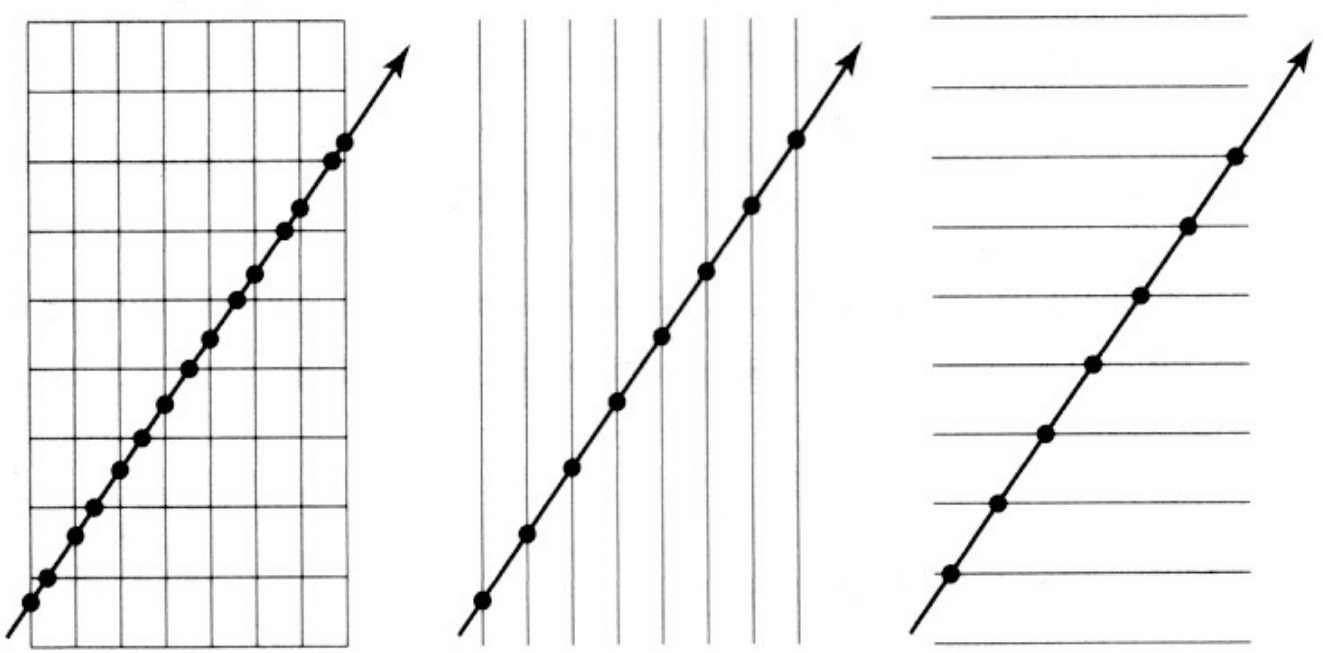


Regular grid example

- Grid divides space, not objects

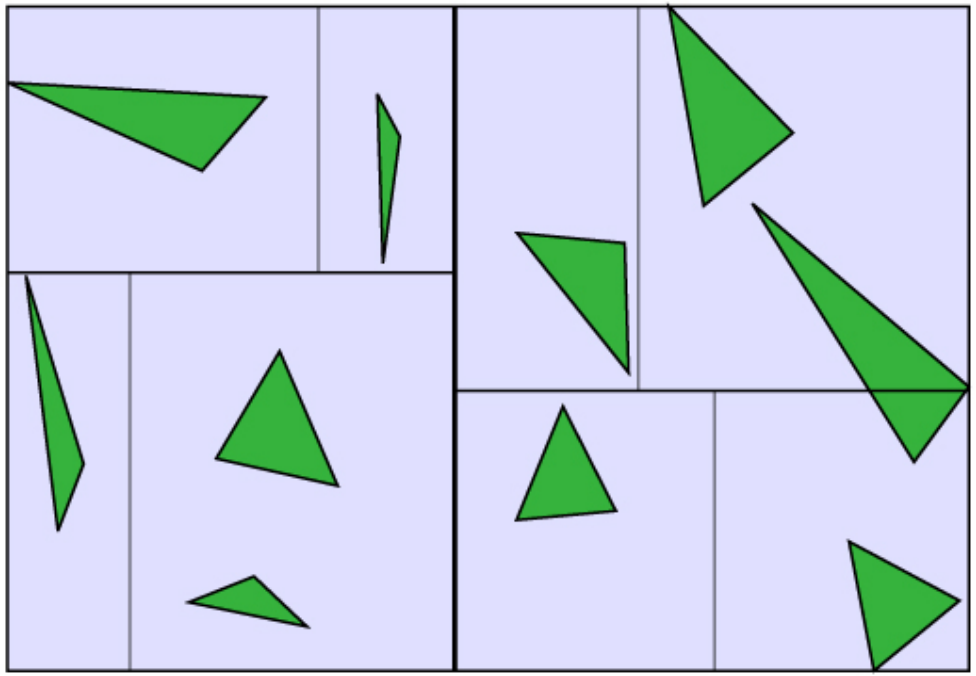


Traversing a regular grid



kd-Tree subdivision

- Subdivides space, like grid, but remains adaptive, like BVH



Practical Acceleration — BVH

- Numerically stable intersection
- Primitives are references only one — bounded memory usage
- Very good intersection performance
- Relatively easy to build, maybe not optimally

