02562 Rendering - Introduction

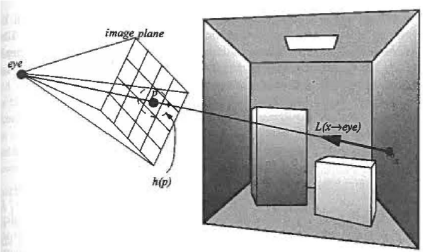
Ray-Object Intersection and Shading Diffuse Surfaces

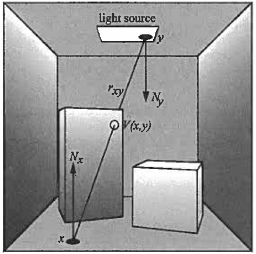
Jeppe Revall Frisvad

September 2023

Rays in theory and in practice

- Parametrisation of a straight line: $\boldsymbol{r}(t) = \boldsymbol{o} + t\vec{\omega}\,, \quad t \in [t_{\mathrm{min}}, t_{\mathrm{max}}]$.
- Camera provides origin ($\boldsymbol{o}$) and direction ($\vec{\omega}$) of "eye rays".





- Rays in code (WGSL) and recording info about a ray-surface intersection:

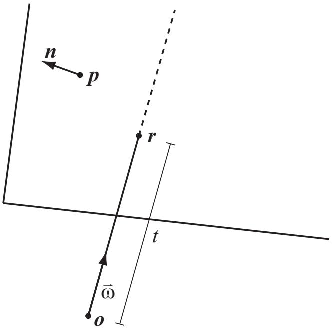
```
struct Ray {
  origin: vec3f,
  direction: vec3f,
  tmin: f32,
  tmax: f32
};
```

```
struct HitInfo {
  has_hit: bool,
  dist: f32,
  position: vec3f,
  normal: vec3f,
  color: vec3f,
  :
};
```

Ray-plane intersection

- Ray: $\boldsymbol{r}(t) = \boldsymbol{o} + t\,\vec{\omega} \quad , \quad t \in [t_{\text{min}}, t_{\text{max}}] \quad .$
- Plane: $ax + by + cz + d = 0 \quad \Leftrightarrow \quad \boldsymbol{p} \cdot \boldsymbol{n} + d = 0\,.$

$d = -\boldsymbol{p}_0 \cdot \boldsymbol{n}$
for some point $\boldsymbol{p}_0$
in the plane.

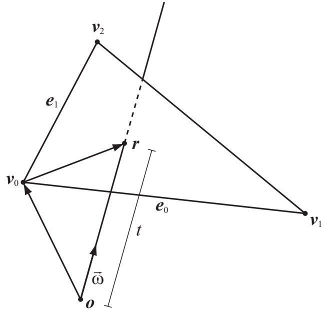


- Setting $\boldsymbol{p} = \boldsymbol{r}$, we find the distance t' to the intersection point:

$$(\boldsymbol{o} + t'\,\vec{\omega}) \cdot \boldsymbol{n} + d = 0 \quad \Leftrightarrow \quad t' = -\frac{\boldsymbol{o} \cdot \boldsymbol{n} + d}{\vec{\omega} \cdot \boldsymbol{n}} = \frac{(\boldsymbol{p}_0 - \boldsymbol{o}) \cdot \boldsymbol{n}}{\vec{\omega} \cdot \boldsymbol{n}}\,.$$

Ray-triangle intersection

- Ray: $\boldsymbol{r}(t) = \boldsymbol{o} + t\,\vec{\omega}, \; t \in [t_{\text{min}}, t_{\text{max}}]$.
- Triangle: $\boldsymbol{v}_0, \; \boldsymbol{v}_1, \; \boldsymbol{v}_2$.



- Edges and normal:
  $\boldsymbol{e}_0 = \boldsymbol{v}_1 - \boldsymbol{v}_0, \; \boldsymbol{e}_1 = \boldsymbol{v}_2 - \boldsymbol{v}_0, \; \boldsymbol{n} = \boldsymbol{e}_0 \times \boldsymbol{e}_1$.
- Barycentric coordinates:

$$\begin{aligned}
\boldsymbol{r}(\alpha,\beta,\gamma) &= \alpha\boldsymbol{v}_0 + \beta\boldsymbol{v}_1 + \gamma\boldsymbol{v}_2 = (1-\beta-\gamma)\boldsymbol{v}_0 + \beta\boldsymbol{v}_1 + \gamma\boldsymbol{v}_2 \\
&= \boldsymbol{v}_0 + \beta\boldsymbol{e}_0 + \gamma\boldsymbol{e}_1 \,.
\end{aligned}$$

- The ray intersects the triangle's plane at $t' = \dfrac{(\boldsymbol{v}_0 - \boldsymbol{o}) \cdot \boldsymbol{n}}{\vec{\omega} \cdot \boldsymbol{n}}$.
- Find $\boldsymbol{r}(t') - \boldsymbol{v}_0$ and decompose it into portions along the edges $\boldsymbol{e}_0$ and $\boldsymbol{e}_1$ to get β and γ. Then check

$$\beta \geq 0 \quad , \quad \gamma \geq 0 \quad , \quad \beta + \gamma \leq 1 \;.$$

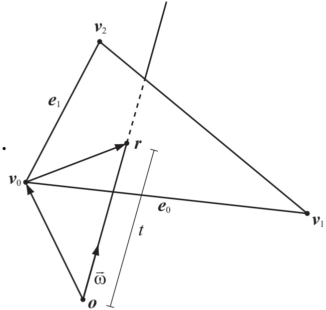
Ray-triangle intersection

- The decomposition of $\boldsymbol{r}(t') - \boldsymbol{v}_0$:

$$\boldsymbol{r}(t') - \boldsymbol{v}_0 = \boldsymbol{o} + t'\vec{\omega} - \boldsymbol{v}_0 = \frac{(\boldsymbol{v}_0 - \boldsymbol{o}) \cdot \boldsymbol{n}}{\vec{\omega} \cdot \boldsymbol{n}}\vec{\omega} - (\boldsymbol{v}_0 - \boldsymbol{o}) \,.$$

- Let us use $\boldsymbol{a} = \boldsymbol{v}_0 - \boldsymbol{o}$, then

$$\begin{aligned}
&(\boldsymbol{r}(t') - \boldsymbol{v}_0)(\vec{\omega} \cdot \boldsymbol{n}) \\
&= \vec{\omega}(\boldsymbol{n} \cdot \boldsymbol{a}) - \boldsymbol{a}(\boldsymbol{n} \cdot \vec{\omega}) = \boldsymbol{n} \times (\vec{\omega} \times \boldsymbol{a}) \\
&= (\boldsymbol{e}_0 \times \boldsymbol{e}_1) \times (\vec{\omega} \times \boldsymbol{a}) = (\boldsymbol{a} \times \vec{\omega}) \times (\boldsymbol{e}_0 \times \boldsymbol{e}_1) \\
&= ((\boldsymbol{a} \times \vec{\omega}) \cdot \boldsymbol{e}_1)\boldsymbol{e}_0 - ((\boldsymbol{a} \times \vec{\omega}) \cdot \boldsymbol{e}_0)\boldsymbol{e}_1 \quad .
\end{aligned}$$



- From this equation, we get the barycentric coordinates of $\boldsymbol{r}(t')$

$$\alpha = 1 - \beta - \gamma \,, \quad \beta = \frac{[(\boldsymbol{v}_0 - \boldsymbol{o}) \times \vec{\omega}] \cdot \boldsymbol{e}_1}{\vec{\omega} \cdot \boldsymbol{n}} \,, \quad \gamma = -\frac{[(\boldsymbol{v}_0 - \boldsymbol{o}) \times \vec{\omega}] \cdot \boldsymbol{e}_0}{\vec{\omega} \cdot \boldsymbol{n}} \,.$$

$\boldsymbol{a} \cdot \boldsymbol{b}$	$=$	$\boldsymbol{b} \cdot \boldsymbol{a}$	(dot product commutation)
$\boldsymbol{a} \times \boldsymbol{b}$	$=$	$-\boldsymbol{b} \times \boldsymbol{a}$	(cross product anticommutation)
$\boldsymbol{a} \cdot (\boldsymbol{b} \times \boldsymbol{c})$	$=$	$(\boldsymbol{a} \times \boldsymbol{b}) \cdot \boldsymbol{c}$	(triple scalar product)
$\boldsymbol{a} \times (\boldsymbol{b} \times \boldsymbol{c})$	$=$	$\boldsymbol{b}(\boldsymbol{a} \cdot \boldsymbol{c}) - \boldsymbol{c}(\boldsymbol{a} \cdot \boldsymbol{b})$	(triple vector product)
$(\boldsymbol{a} \times \boldsymbol{b}) \times (\boldsymbol{c} \times \boldsymbol{d})$	$=$	$((\boldsymbol{a} \times \boldsymbol{b}) \cdot \boldsymbol{d})\boldsymbol{c} - ((\boldsymbol{a} \times \boldsymbol{b}) \cdot \boldsymbol{c})\boldsymbol{d}$	

Ray-sphere intersection

- Ray: $\boldsymbol{r}(t) = \boldsymbol{o} + t\,\vec{\omega},\ \ t \in [t_{\mathrm{min}}, t_{\mathrm{max}}]\,.$
- Sphere: $(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2\,.$
- With $\boldsymbol{p} = (x, y, z)$ and $\boldsymbol{c} = (c_x, c_y, c_z)$, the sphere is

$$(\boldsymbol{p} - \boldsymbol{c}) \cdot (\boldsymbol{p} - \boldsymbol{c}) = r^2\,.$$

- Setting $\boldsymbol{p} = \boldsymbol{r}$, we find the distance t' to the intersection point:

$$(\boldsymbol{o} - \boldsymbol{c} + t'\,\vec{\omega}) \cdot (\boldsymbol{o} - \boldsymbol{c} + t'\,\vec{\omega}) = r^2\,.$$

- This is a second degree polynomial, $at^2 + bt + c = 0$, with

$$a = \vec{\omega} \cdot \vec{\omega} = 1\,, \quad b/2 = (\boldsymbol{o} - \boldsymbol{c}) \cdot \vec{\omega}\,, \quad c = (\boldsymbol{o} - \boldsymbol{c}) \cdot (\boldsymbol{o} - \boldsymbol{c}) - r^2\,.$$

- The distances to the intersection points are

$$t'_1 = -b/2 - \sqrt{(b/2)^2 - c} \quad , \quad t'_2 = -b/2 + \sqrt{(b/2)^2 - c}\,.$$

- There is no intersection if $(b/2)^2 - c < 0$.

The signature of an intersection function and that of a shader

- The intersection function should return a Boolean value signaling intersection or not, but it should also update the hit information when an intersection was found.
- A shader should return an RGB 3-vector result, but it might also change the ray and hit information to prepare for the next ray along the path.
- Using pointers for function arguments:

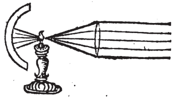
```
fn intersect_plane(r: Ray, hit: ptr<function, HitInfo>, position: vec3f, normal: vec3f) -> bool { ⋯ }
fn intersect_triangle(r: Ray, hit: ptr<function, HitInfo>, v: array<vec3f, 3>) -> bool { ⋯ }
fn intersect_sphere(r: Ray, hit: ptr<function, HitInfo>, center: vec3f, radius: f32) -> bool { ⋯ }
fn lambertian(r: ptr<function, Ray>, hit: ptr<function, HitInfo>) -> vec3f { ⋯ }
fn phong(r: ptr<function, Ray>, hit: ptr<function, HitInfo>) -> vec3f { ⋯ }
fn mirror(r: ptr<function, Ray>, hit: ptr<function, HitInfo>) -> vec3f { ⋯ }

fn shade(r: ptr<function, Ray>, hit: ptr<function, HitInfo>) -> vec3f
{
  switch (*hit).shader {
    case 1 { return lambertian(r, hit); }
    case 2 { return phong(r, hit); }
    case 3 { return mirror(r, hit); }
    ⋮
    case default { return (*hit).color; }
  }
}
```

Kepler's inverse square law

As the relation of a spherical surface, which has its centre in the origin of the light, is from a larger to a smaller one: such is the relation of the strength or density of light rays in a smaller to that in a more spacious spherical surface, that is, conversely. [Kepler 1604]

- Light from a point source falls off with the square of the distance to the point.
- Kepler struggled with this law since he only had the notion of rays of light (light was not considered to spread in a volume).
- Neither did he have a precise law of refraction, but he did observe that the inverse square law only works for point lights.
  He did this by generating collimated/directional light using concave mirrors and convex lenses.



- If a point light at $\boldsymbol{p}$ has intensity I, the light incident at a surface point $\boldsymbol{x}$ is

$$L_i = \frac{I}{\|\boldsymbol{p} - \boldsymbol{x}\|^2} \ .$$

Lambert's cosine law

brightness decreases in the same ratio by which the sine of the angle of incidence decreases [Lambert 1760]

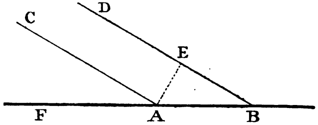


Fig. 1.

- Lambert uses the angle $(\mathrm{CAF} = \mathrm{DBF})$ between the direction of the rays $(\mathrm{CA}$ and $\mathrm{DB})$ and the surface tangent plane (AB) as the angle of incidence.
- If we instead measure the angle of incidence θ from the normalised surface normal $\vec{n}$ to the direction toward the incident light $\vec{\omega}_i$, sine becomes cosine.
- Then the diffusely reflected light is

$$L_{r,d} = \frac{\rho_d}{\pi} L_i \cos\theta = \frac{\rho_d}{\pi} L_i \left(\vec{n} \cdot \vec{\omega}_i\right) \ .$$

where ρ_d is the diffuse reflectance.

Sampling light in the shader

- The shaders job is to compute and return the amount of observed light $L_o = L_e + L_r$, where L_e is emitted light and L_r is reflected light.
- The reflected light L_r is light reflected directly from the light source and also light due to indirect illumination.
- For approximation of a diffuse surface, we can use

$$L_o = L_e + L_r = L_e + L_{r,d} + L_a = L_e + \frac{\rho_d}{\pi} L_i \left(\vec{n} \cdot \vec{\omega}_i\right) + L_a \ ,$$

  where L_a approximates the indirect illumination by a constant (this is sometimes called *ambient* light).
- To get the incident light L_i, we sample a light source. For each type of light source, we should have a sample function returning a `Light` struct.

```
struct Light {
  L_i: vec3f,
  w_i: vec3f,
  dist: f32
};
```

```
fn sample_point_light(pos: vec3f) -> Light
{
  ⋮
}
```

Intersecting with the scene and tracing a path

- If no recursion, the main program of the ray tracer uses a loop to trace a path:

```
@fragment
fn main_fs(@location(0) coords: vec2f) -> @location(0) vec4f
{
  const bgcolor = vec4f(0.1, 0.3, 0.6, 1.0);
  const max_depth = 10;
  let uv = vec2f(coords.x*uniforms.aspect*0.5f, coords.y*0.5f);
  var r = get_camera_ray(uv);
  var result = vec3f(0.0);
  var hit = HitInfo(false, 0.0, vec3f(0.0), vec3f(0.0), vec3f(0.0), … );
  for(var i = 0; i < max_depth; i++) {
    if(intersect_scene(&r, &hit)) { result += shade(&r, &hit); }
    else { result += bgcolor.rgb; break; }
    if(hit.has_hit) { break; }
  }
  return vec4f(pow(result, vec3f(1.0/uniforms.gamma)), bgcolor.a);
}
```

- The intersect scene function contains the specifics of the scene description and calls the intersection functions for the different objects in the scene:

```
fn intersect_scene(r: ptr<function, Ray>, hit : ptr<function, HitInfo>) -> bool
{
  // Define scene data as constants.

  // Call an intersection function for each object.
  // For each intersection found, update (*r).tmax and store additional info about the hit.

  return (*hit).has_hit;
}
```

Exercises

Week 1

- Implement a render engine based on WebGPU.
- Generate rays using a (modified) pinhole camera model.
- Pass data from CPU to GPU to enable interaction and load balancing.

Week 2

- Compute ray-plane, ray-triangle, and ray-sphere intersection.
- Implement conditional acceptance of intersections to find the closest hit (hidden surface removal).
- Compute shading of diffuse surfaces by point lights.
  Use Kepler's inverse square law and Lambert's cosine law.