Tutorial 7

Illumination

Computer Graphics

Summer Semester 2020
Ludwig-Maximilians-Universität München
Agenda

- Whitted-style Ray Tracing
- Monte Carlo Ray Tracing
  - Monte Carlo Integration
  - Path Tracing
  - Light Sampling
  - Direct & Indirect Illumination
- Epilogue
Tutorial 7: Illumination

- Whitted-style Ray Tracing
- Monte Carlo Ray Tracing
  - Monte Carlo Integration
  - Path Tracing
  - Light Sampling
  - Direct & Indirect Illumination
- Epilogue
Revisit: Phong and Blinn-Phong's (Local) Model

\[ L_{\text{Phong}} = L_a + L_d + L_s = k_a I_a + k_d I_d \max (0, \mathbf{N} \cdot \mathbf{L}) + k_s I_s \max (0, \mathbf{R} \cdot \mathbf{V})^p \]

\[ L_{\text{Blinn-Phong}} = L_a + L_d + L'_s = k_a I_a + k_d I_d \max (0, \mathbf{N} \cdot \mathbf{L}) + k_s I_s \max (0, \mathbf{N} \cdot \mathbf{H})^p \]

What do we do about reflected light?

What do we do about shadows in these models?

…

Intuitively, they totally do not match the real world!
Whitted-style: An Improved (Global) Illumination Model

Simple idea: *Ray tracing*. Trace a ray’s path, sum up the intensity

\[
L_{\text{Whitted}} = k_a L_a + k_d I_d \mathbf{N} \cdot \mathbf{L} + k_s S + k_t T
\]

- **Ambient Term**
- **Diffuse Term**
- **Whitted Term**

Intensity from a reflected ray

Intensity from a transmission ray

Whitted-style Ray Tracing (1980)

- Always perform specular reflections / refractions
- Stop bouncing at diffuse surface
Ray Tracing in Practice: Performance Issue

● Naive way to trace a ray
  ○ Exhaustively test ray intersections with every object (each object has many triangles)
  ○ Which object will be intersected?
  ○ Which triangle will be hit?
  ○ What are the coordinates of the hit position?
  ○ When a ray is reflected/refracted, should the intersection of all objects be considered for the new ray?
  ○ …

● Performance issue:
  ○ Naive ray tracing = #pixels x #triangles x #bounces
  ○ This is really slow, and why ray tracing is so hard
This Scene is Rendered using Ray Tracing (in Real-Time)

- Each statue has more than 33 million triangles
- Naive ray tracing would not work here in any cases
- A clever acceleration structure is needed for sure
Can we fake it with Rasterization (in Real-Time)?

Glass: Refraction Effects

Metal: Reflection Effects

Shadows
Task 1 a) Add Checkerboard

```javascript
constructor() {

    // TODO: create a checkerboard with red and yellow color
    const g = new PlaneGeometry(
        this.params.plane_width, this.params.plane_height,
        this.params.plane_width, this.params.plane_height
    )

    for (let j = 0; j < this.params.plane_height*2; j+=2) {
        for (let i = 0; i < this.params.plane_width*2; i+=2) {
            g.faces[this.params.plane_height*2*j + i].materialIndex = (i/2+j/2) % 2 === 0 ? 0 : 1
            g.faces[this.params.plane_height*2*j + i+1].materialIndex = (i/2+j/2) % 2 === 0 ? 0 : 1
        }
    }

    this.p = new Mesh(g, [
        new MeshPhongMaterial({color: this.params.color_plane[0]}),
        new MeshPhongMaterial({color: this.params.color_plane[1]})
    ]) this.p.position.copy(this.params.position.plane)
    this.p.rotateX(-Math.PI/2)
    this.scene.add(this.p)

    // enable shadows
    this.enableShadow()
}
```
Task 1 a) Add Glass Sphere

```javascript
constructor() {
    // TODO: create a glass sphere based on MeshBasicMaterial.
    this.refractionCamera = new CubeCamera(0.1, 5000, 512)
    this.refractionCamera.renderTarget.mapping = CubeRefractionMapping
    this.scene.add(this.refractionCamera)
    this.s1 = new Mesh(
        new SphereGeometry(this.params.radius, 100, 100),
        new MeshBasicMaterial({
            color: 0xffffff,
            envMap: this.refractionCamera.renderTarget,
            side: BackSide,
            refractionRatio: this.params.refractionRatio,
            reflectivity: this.params.reflectivity
        }),
    )
    this.s1.position.copy(this.params.position.right)
    this.refractionCamera.position.copy(this.s1.position)
    this.scene.add(this.s1)
    ...
}
update() {
    // TODO: implement update if you needed (yes we need it).
    this.s1.visible = false
    this.refractionCamera.update(this.renderer, this.scene)
    this.s1.visible = true
}
```
Task 1 a) Add Metal Sphere and Shadows

```
constructor() {
  ...
  // TODO: create a metal sphere based on phong material
  this.s2 = new Mesh(
    new SphereGeometry(this.params.radius, 100, 100),
    new MeshPhongMaterial({color: this.params.color_sphere, side: FrontSide}),
  )
  this.s2.position.copy(this.params.position.left)
  this.scene.add(this.s2)
  ...
}

enableShadow() {
  // TODO: enable shadows for objects you have created
  this.renderer.shadowMap.enabled = true
  this.renderer.shadowMap.type = PCFSoftShadowMap
  this.s1.castShadow = true
  this.s1.receiveShadow = true
  this.s2.castShadow = true
  this.s2.receiveShadow = true
  this.p.receiveShadow = true
}```
A Fake Whitted-style

Live Demo: https://www.medien.ifi.lmu.de/lehre/ss20/cg1/demo/7-illumination/whitted/index.html
Other Possible Solutions

The demonstrated solution is not perfect, you could also come up with other solutions using:

- Customized shaders
- Texture mapping
- …

Be more creative :)
Task 1 b) What went wrong here?

- No soft shadow
- No caustics effect on glass sphere
- No reflection on the metal sphere
- Cast shadow on the metal sphere does not appear on the glass sphere
- Shadow resolution is low here
Task 1 c) More physically-based Effects?

- Absorption
- Scattering
- Polarization
- Diffraction
- Interference
- …

None of them are possible with Whitted-style ray tracing.
Revisit: The Rendering Equation (1986)

\[ L_o(x, w_o) = L_e(x, w_o) + L_r(x, w_o) = L_e(x, w_o) + \int_{\Omega} f_r(x, w_i, w_o) L_i(x, w_i) \cos \theta_i \, dw_i \]

Task 1 d) Whitted-style is "wrong"

The rendering equation integrates the radiance from all incoming directions (hemisphere) but the whitted style ray tracing does not consider other indirect effects (single ray path).

The rendering equation is "correct", at least more correct than Whitted-style.

But how to solve it?
Tutorial 7: Illumination

- Whitted-style Ray Tracing
- Monte Carlo Ray Tracing
  - Monte Carlo Integration
  - Path Tracing
  - Light Sampling
  - Direct & Indirect Illumination
- Epilogue
Revisit: Probability and Statistics

- (Continuous) random variables \( X \sim p(x) \)
- Probability \( P(a \leq X \leq b) = \int_a^b p(x)\,dx \)
- Cumulative distribution function (CDF) \( P(X \leq x) = \int_{-\infty}^x p(t)\,dt \)
- Probability density function (PDF) \( p(x) \), e.g. Gaussian
- Expected value \( E(x) = \int_{-\infty}^{\infty} xp(x)\,dx \)

https://en.wikipedia.org/wiki/Normal_distribution
Monte Carlo Integration

It can be too hard to solve a definite integration analytically. The Monte Carlo Integration is a method to estimate the definite integral of a function by averaging random samples of the function's value.

The rendering equation can be solved using the Monte Carlo integration:

\[
\int_{a}^{b} f(x) \, dx = \frac{1}{N} \sum_{k=1}^{N} \frac{f(X_k)}{p(X_k)}, \quad X_k \sim p(x)
\]

The rendering equation using Monte Carlo integration is:

\[
L_o(x, w_o) = L_e(x, w_o) + \frac{1}{N} \sum_{k=1}^{N} \frac{L_{i,k}(x, w_{i,k}) f_r(x, w_{i,k}, w_o,k) \cos \theta_{i,k}}{p(w_{i,k})}
\]

https://en.wikipedia.org/wiki/Monte_Carlo_integration
Task 2 a) The Rendering Equation with Lambertian BRDF

In Assignment 6 - Task 3 a), we derived the BRDF $f_r = \frac{1}{\pi}$ for a Lambertian surface/material. Note that the Lambertian material only takes into account diffuse incoming radiance, and doesn't absorb or emit radiance, i.e. $L_e(x, w_o) = 0$

So the rendering equation using Monte Carlo integration is:

$$L_o(x, w_o) = \frac{1}{N} \sum_{k=1}^{N} \frac{L_{i,k}(x, w_{i,k}) \cos \theta_{i,k}}{\pi p(w_{i,k})}$$
Task 2 b) Uniform Sampling on Hemisphere

Uniform sampling means the density function is a constant, thus \( p(w_{i,k}) = c \).

Due to the definition of the probability density function (PDF), we have:

\[
\int_{\Omega} p(w_{i,k}) \, dw_{i,k} = c \int_{\Omega} \, dw_{i,k} = 1
\]

The definite integral is on a hemisphere, with the spherical coordinates:

\[
\int_{\Omega} \, dw_{i,k} = \int_{\theta=0}^{\pi/2} \int_{\phi=0}^{2\pi} \, d\theta \sin \theta \, d\phi
\]

\[
= \int_{0}^{\pi/2} \sin \theta \, d\theta \int_{0}^{2\pi} \, d\phi
\]

\[
= (-\cos \frac{\pi}{2} + \cos 0)(2\pi - 0) = 2\pi
\]

Therefore \( p_{\text{hemi}}(w) = p(w_{i,k}) = \frac{1}{2\pi} \) (i.e. 1 / surface area)
Solving The Rendering Equation (The Easy/Naive Case)

The rendering equation on a Lambertian surface, with uniform sampling on a hemisphere can be simplified to:

\[ L_o(x, w_o) = L_e(x, w_o) + \int_{\Omega} f_r(x, w_i, w_o) L_i(x, w_i) \cos \theta_i \, dw_i = \frac{2}{N} \sum_{k=1}^{N} L_{i,k}(x, w_{i,k}) \cos \theta_{i,k} \]

A simple Monte Carlo estimation for the rendering equation (pseudocode):

```c
// x is shading point, wo is the outgoing ray

shade(x, wo, bounces) {
    Lo = 0
    if bounces == 0 { return hit light ? light_emission : 0 } // termination condition of recursive function call
    for each randomly sample N directions wi {
        trace a ray
        if ray hit the light: Lo += light_emission * cosine_theta
        if ray hit object at q: Lo += shade(q, -wi, bounces-1) * cosine_theta // f_r = 1/pi, pdf(wi) = 1/2pi
    }
    return 2*Lo/N
}
```

Recursive depth == #bounces, thus #rays = N#bounces ⇒ explode!
When N=1: Path tracing, just a single ray
When N>1: Distributed ray tracing, tracing exploded #rays
Global Illumination via Path Tracing

Sampling multiple random directions can cause #ray to explode

But we can shoot multiple ray paths from a pixel to gain linear complexity

i.e. multiple samples per pixel (spp)
Global Illumination via Path Tracing (cont.)

// x is shading point, wo is the outgoing ray
shade(x, wo, bounces) {
    Lo = 0
    if bounces == 0 { return hit light ? light_emission : 0 }
    randomly sample ONE direction wi
    trace a ray
    if ray hit the light: Lo += light_emission * cosine_theta
    if ray hit object at q: Lo += shade(q, -wi, bounces-1) * cosine_theta // f_r = 1/pi, pdf(wi) = 1/2pi
    return 2*Lo/N
}

What can we do about this?

What if we running out of the #bounces and could not hit the light? The whole recursion will get 0 ⇒ inefficient. A lot of path are "wasted"
Task 2 c) Sampling Area Light (Direct Illumination)

Recall the definition of a solid angle (it is the projected area on the unit sphere)

We can express $dw$ by $ds$ immediately using the definition:

$$dw = \frac{\cos \theta' ds}{|x - x'|^2}$$
The Rendering Equation by Sampling Light Source

In this case, the PDF is $1/S$. The rendering equation is

$$L_o(x, w_o) \approx L_e(x, w_o) + \int_S f_r(x, w_i, w_o) L_i(x, w_i) \frac{\cos \theta_i \cos \theta'}{|x - x'|^2} \, ds$$

The corresponding Monte Carlo solution that samples an area light:

$$L_o(x, w_o) = \frac{1}{N} \sum_{k=1}^{N} \frac{L_{i,k}(x, w_i, k) \cos \theta_{i,k} \cos \theta'_{i,k}}{\pi \frac{1}{S} |x - x'|^2} = \frac{1}{N} \sum_{k=1}^{N} \frac{L_{i,k}(x, w_i, k) \cos \theta_{i,k} \cos \theta'_{i,k} S}{\pi |x - x'|^2}$$
Task 2 d) Implementing A Simple Path Tracer

In the lecture, you learned that WebGL 2.0 does not support recursion, thus we need to turn the accumulation to a non-recursive version. The pseudocode from Assignment 7 already tells you how to do it:

```c
shade(wo) {
    Li = 0
    Lo = 0
    for i = 0; i < bounces; i++ {
        if wo not hit the world {
            return Lo
        }
        if wo hit light source {
            return Lo
        }
        Li = Li * hitted material color
        if light is not blocked in the middle {
            Lo += radiance at hit position // use the rendering equation
        }
        wo = randomly sample one direction
    }
    return Lo
}
```

\[ L_{i,k}(x, w_{i,k}) \cos \theta_{i,k} \cos \theta'_{i,k} S \]
\[ \pi |x - x'|^2 \]
Task 2 d) Implementing A Simple Path Tracer (cont.)

```c
vec3 shade(in ray r) {
    vec3 Li = vec3(0); vec3 Lo = vec3(0); hit_record hit;
    float pdf_light = 1.0 / area_light_surface; // 1/S
    float f_r = 1.0 / pi; // lambertian
    // TODO: implement path tracing, return the emitted color of the given ray.
    for (int i = 0; i < 1+bounces; i++) {
        if (!world_hit(r, hit)) { return Lo; }
        if (hit.mat.type == light_source) { return i == 0 ? vec3(light.color) : Lo; }
        vec3 Lpos = area_light.center + area_light.dimension * (2.*random3()-1.);
        vec3 l = Lpos - hit.p;
        float distance_sequare = dot(l, l); // |x-x'|^2
        l = normalize(l);
        ray shadow_ray = ray(hit.p, l);
        Li = i == 0 ? hit.mat.color : Li * hit.mat.color;
        float cosine_theta1 = dot(hit.normal, l); // cosθ
        float cosine_theta2 = dot(area_light_normal, -l); // cosθ'
        if (cosine_theta1 > 0.0 && cosine_theta2 > 0.0 && !shadow_hit(shadow_ray)) { // direct illumination
            Lo += Li * light.color * f_r * cosine_theta1 * cosine_theta2 / distance_sequare / pdf_light;
        }
    }
    return Lo;
}
```

- 14 lines of code (can be shorter by remove intermediate variables)
- Indeed difficult to understand from beginning to the end
- Even not easy to debug
- But you can get a huge sense of accomplishment for sure if you did it right
Math and physics worked!
Task 2 e) Changing Light Bounces

- Increasing light bounces introduces more indirect illumination to non emission materials.
- Infinite light bounces won't make the scene become pure white as how real world light behaves (or energy conservation: reflected radiance $\leq$ incoming radiance).

But how to decide the number of light bounces?
Biased vs. Unbiased Monte Carlo Estimator

- Unbiased: No systematic error, or the expected value of the Monte Carlo estimator is equal to the definite integral
- Biased: otherwise

Without infinite light bounces, our calculated color is always biased.

What can we do about this?
Solution: Russian Roulette (Using Bernoulli Distribution)

Assume we don't set a limit to the number of light bounces, instead of manually giving a probability $p$ ($0 < p < 1$)

- With a probability $p$, we keep shooting (reflect) an accumulate ray at the Monte Carlo integration, the accumulate radiance is divided by $p$: $L_i / p$

- With a probability $1-p$, we terminate bouncing the ray $\Rightarrow$ returns radiance: 0

The expected value (i.e. Bernoulli 0-1 distribution):

$$E = p \cdot \sum_i \frac{L_i}{p} + (1 - p) \cdot 0 = \sum_i L_i = L_o$$

Now we don't rely on the number of bounces and have an unbiased estimation (even doesn't depend on the probability you have chosen anymore)
Path Tracing (unbiased complete version, pseudocode)

// x is shading point, wo is the outgoing ray
shade(x, wo) {
    // direct illumination, contribution from the light source
    Ldir = 0
    ray = x - x' // x' is from light source
    if ray is not blocked in the middle {
        Ldir = light_color * f_r * cosθ * cosθ' / |x'-x|^2 / pdf_light // integrate over light source
    }

    // indirect illumination, contribution from non-emitting materials
    Lindir = 0
    if random() < p_rr { // p_rr is the given russian roulette probability
        wi = sample(wo, n) // hemisphere sampling ONE incoming direction for the next ray
        if wi hit non emitting object at q {
            Lindir = shade(q, -wi) * f_r * cosθ / pdf_hemi / p_rr // task 2 b): pdf_hemi = 1/2pi
        }
    }
    return Ldir + Lindir
}

Path Tracing (unbiased complete version, pseudocode)
Task 2 f) Changing spp

- Increasing spp can reduce the sampling noise
- Because of the law of large numbers, we need infinite samples per pixel eventually

But what can we do about it?
Denoising!

- An appropriate denoising approach can reduce the samples per pixel that we need.
- Ideally, we want a denoised image that rendered with 1 spp (why?)
Noise Reduction for Cornell Box

This cornell box is created using Blender, and denoised by Blender's built-in denoiser.
Tutorial 7: Illumination

● Whitted-style Ray Tracing

● Monte Carlo Ray Tracing
  ○ Monte Carlo Integration
  ○ Path Tracing
  ○ Light Sampling
  ○ Direct & Indirect Illumination

● Epilogue
Topics not covered in detail (or where to go from here)

- Acceleration structure: find the right object
  - partially covered, see BVH in the Rasterization; volume rendering, see lecture slides

- Ray casting: find the right position
  - partially covered, see Liang's algorithm in the Rasterization, or read code in cornell box task carefully

- Importance sampling
  - advanced(?) math techniques for solving Monte Carlo integration more efficiently

- Noise reduction
  - advanced(?) math techniques for solving Monte Carlo integration with fewer spp ⇒ Real-time ray tracing!

- Global illumination approximation
  - calculate the rendering equation for different geometric structures (e.g. voxel) ⇒ Real-time ray tracing!

- Light transportation variances
  - More physical concern, e.g. photon mapping, metropolis light transport ⇒ Precise accurate offline rendering
Rasterization v.s. Ray Tracing in Real-Time

Shaders in the pipeline:
- **Vertex shader**: transforming vertices
- **Tessellation shader**: subdividing meshes
- **Geometry shader**: generating new primitives
- **Fragment shader**: colorizing pixels

Shader not in the pipeline:
- **Compute shader**: for general purpose computing

```plaintext
init frame buffer
init z buffer
for each triangle t in scene {
  tp = project(t)
  for each pixel p in frame buffer {
    if tp covers p {
      if z value at p is closer than z buffer at p {
        update z buffer and frame buffer
      }
    }
  }
} flush frame buffer to monitor
```

**Memory**
- Uniforms
- Textures
- Buffers
- ...

*This pipeline is standardized in OpenGL, NVIDIA's hardware supports an implementation*
Rasterization v.s. Ray Tracing in Real-Time

Shaders in the pipeline:
- **Raygen shader**: deal with rays and write final output to memory
- **Intersection shader**: handling ray-primitives intersection
- **Closest-hit shader**: only on the closest hit position
- **Any-hit shader**: for all possible intersections
- **Rmiss shader**: invoke when no intersection is found

```plaintext
init frame buffer
for each pixel p in frame buffer {
    construct a ray from p
    for ray bounces is not over {
        for each triangle t in the scene {
            if ray hit t at x {
                keep x if closest and update the ray
                break
            }
        }
    }
    update frame buffer
}
flush frame buffer to monitor
```

*This pipeline is proposed by NVIDIA, implemented in RTX-series hardwares*
Real-Time Ray Tracing (RTRT) Today

● Real-time rendering related research advances
  ○ Parallelized BVH construction and traversal algorithm research (approx. 2010-2013)
  ○ Voxel-based global illumination research (approx. 2005-2012)
  ○ …

● Industrial practices
  ○ NVIDIA's RTX hardware implementation (2018-today)
    ■ Works for PCs
  ○ Epic Games' UE5 software implementation (2020-today)
    ■ Works for PlayStation 5, XBOX, …
Take Away

- Path tracing is old fashioned (1986) but still the mostly used global illumination solution for industrial photorealistic rendering, and have been largely applied for decades!
- Too many exciting advances which the course is too basic/short to contain :)
- RTRT is replacing (?) rasterization over the next generation (decade)
- We don't know the future, but what do you think?
- Check these books if you still interested in CG and want dive further:
Thanks!
What are your questions?