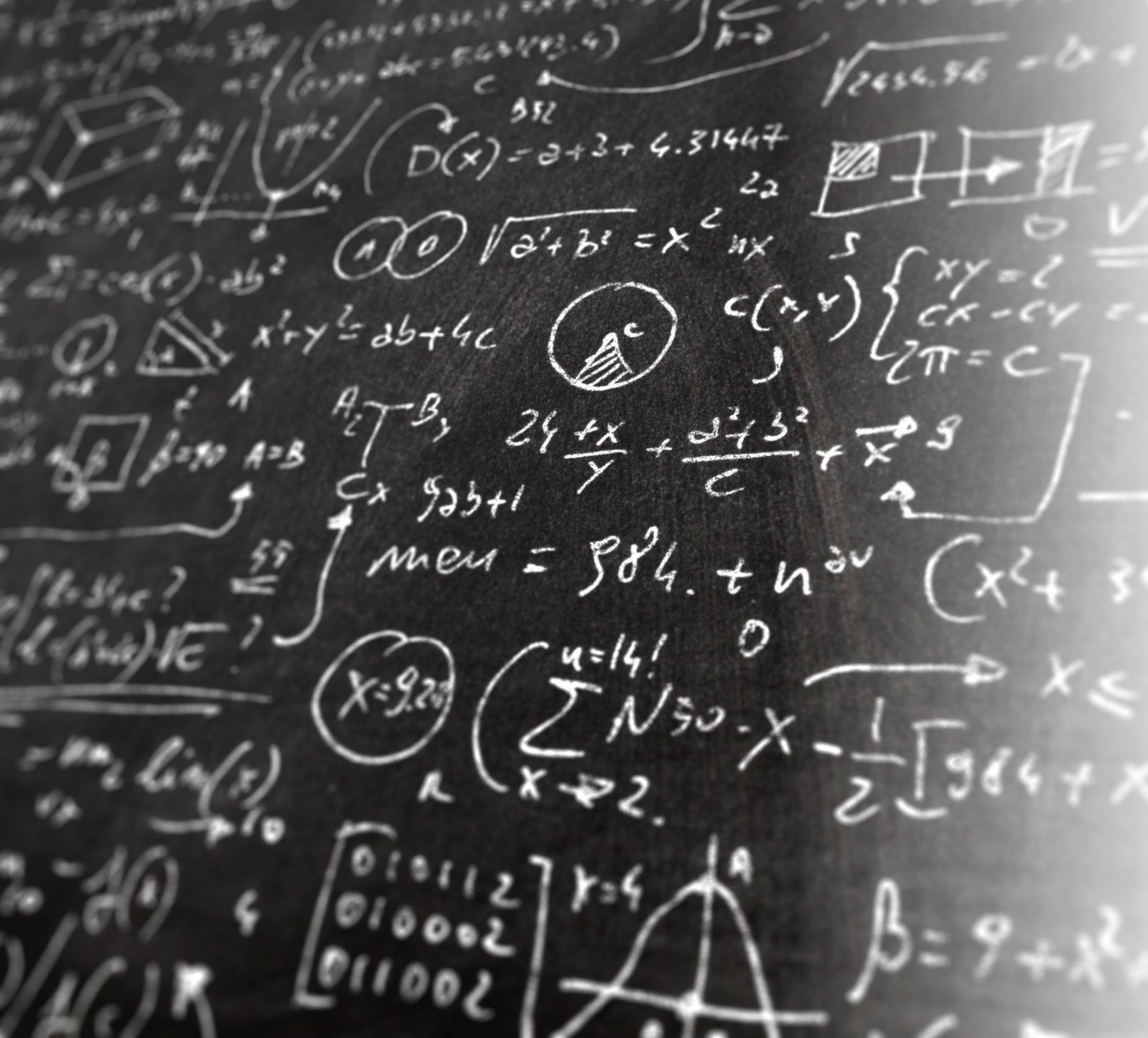
Raytracing

Dr. Rafael Kuffner dos Anjos

Agenda

- Rendering Equation
- Implementation steps



Rendering Equation

- The answer according to physics
- A single equation that captures all light
- We will then have to convert it to code
- But the idea is pretty simple
- We will, however, need some assumptions

Assumptions

- Initially, assume reflection only
  - i.e. scattering is defined by a BRDF f_r
  - broken in two parts
    - impulse (mirror) reflection/transmission
    - scattering via integration
- Point (impulse) and area (integral) luminaires
- Scene is finite and constructed of manifolds

More Assumptions

- Escaping light disappears (black sphere)
- Ray-casting function $R: \mathcal{M} \times \boldsymbol{S}^{\mathbf{2}} \to \mathcal{M}$
  - finds first intersection Q along ray (P, d)
- Steady-state (no time dependence)
- Radiance only (no wavelength dependence)
- Emitted radiance at all points:
  - $L^e: \mathcal{M} \times \boldsymbol{S}^{\mathbf{2}} \to \mathbb{R}: \ (P, \omega) \to L^e(P, \omega)$

Rendering Equation

- $L^{ref}(P, \omega_0) = \int_{\omega_i \in \boldsymbol{S}^{\boldsymbol{2}}_{+}(P)} L(P, -\omega_i) f_r(P, \omega_i, \omega_0) \left(\omega_i \cdot \overrightarrow{n_p}\right) \mathrm{d}\omega_i$
- $L^{ref}(P, \omega_0)$: Light reflected at P in direction ω_0
- $\int_{\omega_i \in \boldsymbol{S}^{\boldsymbol{2}}_{+}(P)} \dots \mathrm{d}\omega_i$: Integral over all incoming directions ω_i
- $L(P, -\omega_i)$: Light flow coming *in* from ω_i
- $f_r(P, \omega_i, \omega_0)$: BRDF: reflectance from ω_i to ω_o
- $\omega_i \cdot \overrightarrow{n_p}$: Dot product for angled patch

UNIVERSITY OF LEEDS

Adding Emission

- $L(P, \omega_0) = L^e(P, \omega_0) + L^{ref}$
  - $= L^e(P, \omega_0) + \int_{\omega_i \in \boldsymbol{S}_+^2(P)} L(P, -\omega_i) f_r(P, \omega_i, \omega_0) \left(\omega_i \cdot \overrightarrow{n_p}\right) \mathrm{d}\omega_i$
- $L^e(P, \omega_0)$ is the surface radiance
- $L^{ref}(P, \omega_0)$ is the field radiance
- Originally described by Kajiya and by Immel in 1986
- This is an integral equation
  - Effectively impossible to solve analytically

Transport Equation

- Field radiance has to come from somewhere
- So trace back along each direction
- It must be the *outgoing* light from somewhere
  - $L(P, -\omega_i) = L(R(P, \omega_i), -\omega_i)$
- Results in:
- $L(P, \omega_0) = L^e(P, \omega_0) + \int_{\omega_i \in \boldsymbol{S}_+^2(P)} L(R(P, \omega_i), -\omega_i) f_r(P, \omega_i, \omega_0) \left(\omega_i \cdot \overrightarrow{n_p}\right) \mathrm{d}\omega_i$

Solving Computationally

- For each ray (pixel)
  - Find first intersection along ray
  - On the first bounce, add emission
  - Add direct light from point (impulse) lights
  - Add direct light from area lights
  - Add indirect light from reflections

Surface Element Class

```
// a surface element with accessor functions for normals, &c.
// name is short for "Surface Element"
Surfel
    Surface Owner
    Point Position
    TexCoord UV
    Vector Normal
    float Emission
    float AmbientAlbedo
    float LambertAlbedo
    float GlossyAlbedo
    float GlossyExponent
    float BRDF(Vector InDirection, Vector OutDirection)

// a BRDF function that implements Blinn-Phong
float Surfel::BRDF(Vector OutDirection, Vector InDirection)
    // compute Lambertian aka diffuse
    L = normal.angleCosine(InDirection)
    // compute glossy
    G = pow(normal.angleCosine((OutDirection+InDirection)/2), exponent)
    // sum them and return
    return L * LambertAlbedo + G * GlossyAlbedo
```

A Simple Raytracer

```
// a simple Ray class with a suitable constructor
Ray(Point Origin, Vector Direction)

// SIMPLE BLINN-PHONG RAYTRACER
// loop through all pixels
for each pixel p_ij
    image[i][j] = 0
// loop through all pixels
for each pixel p_ij
    // add the radiance to the pixel
    // note that Ray's first parameter is the origin of the ray
    // but the destination of the light
    image[i][j] += pathTrace(Ray(Eye, p_ij - Eye))
```

Path Trace Function

```
// path tracing routine
float pathTrace(Ray R)
    // total radiance to return
    totalRadiance = 0
    // compute intersection point
    Triangle T = ClosestTriangleAlong(R)
    Surfel S = Intersection(R, T)
    // estimate lighting
    totalRadiance += S.Emission
    // now loop through light sources
    for (each Light)
        // add direct reflection for that light source
        totalRadiance += DirectLight(S, -R.Direction, Light)
    // add indirect light
    totalRadiance += IndirectLight(S, -R.Direction)
    // return the result
    return totalRadiance
```

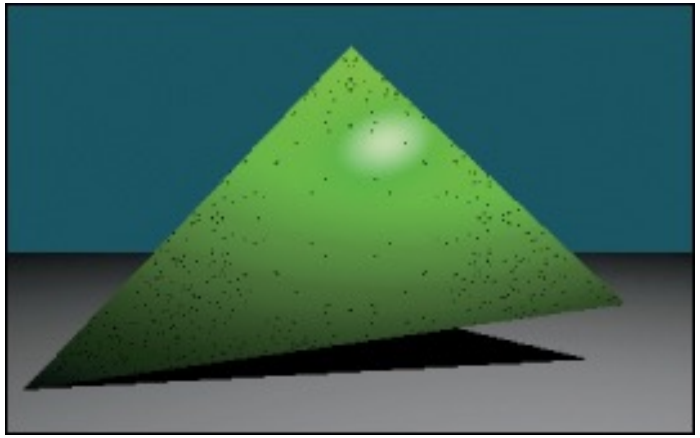
Lighting Functions

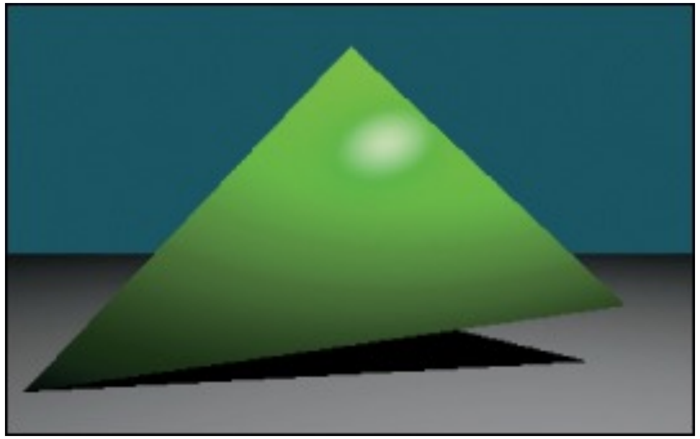
```
// computation for direct light
float DirectLight(Surfel S, Vector OutDirection, Luminaire Light)
    // find incoming light direction
    InDirection = Light.Position - S.Position
    // call BRDF to find total albedo. multiply & return
    return Light.Intensity * S.BRDF(OutDirection, InDirection)

// computation for indirect light
float IndirectLight(Surfel S, Vector OutDirection)
    // use the ambient term to estimate
    return Light.Intensity * S.AmbientAlbedo
```

Intersection Problems

- Intersection is a numerical test
- So we might be off by a small epsilon
- This causes issues with lighting
- Solution: displace intersection
  - Small amount along normal vector
  - Assume this is done in the intersection code





Upgrading Direct Lighting

- Test shadow ray from P to light source
  - If it intersects an object, no direct light
- Add attenuation
  - Infinite light sources do not attenuate
    - Set attenuation to 1.0
  - Finite light sources use inverse square law
    - Set attenuation to distance squared

Shadows & Attenuation

```
// SHADOWS & ATTENUATION
float DirectLight(Surfel S, Vector OutDirection, Luminaire Light)
    // find incoming light direction
    InDirection = Light.Position - P
    // now test shadow ray
    T = ClosestTriangleAlong(InDirection)
    if (T exists)
        return 0

    // now work out attenuation due to distance squared
    // infinite lights effectively have no attenuation
    if (Light.AtInfinity)
        dSqr = 1
    else
        dSqr = InDirection.dot(InDirection)
    // scale by BRDF & light intensity, then return
    return Light.Intensity * S.BRDF(OutDirection, InDirection) / dSqr
```

Indirect Lighting

$$\int_{\omega_i \in \boldsymbol{S}^2_+(P)} L(P, -\omega_i) f_r(P, \omega_i, \omega_0)\left(\omega_i \cdot \overrightarrow{n_p}\right) \mathrm{d}\omega_i$$

- Assume all remaining light is indirect
- Integrate over *all* incoming directions
- Practical solution:
  - Average over many incoming directions
  - Chosen randomly (whatever that means)
  - Find outgoing light from previous bounce

Inefficient Integration

```
// INEFFICIENT INDIRECT LIGHT
// fails due to infinite recursion
float IndirectLight(Surfel S, Vector OutDirection)
    // start with no radiance
    totalRadiance = 0
    // and integrate over the incoming hemisphere
    for (n_samples iterations)
        // create a vector for incoming direction
        InDirection = EvenlyDistributedVector(S.Normal)
        // find the incoming irradiance
        InLight = pathTrace(Ray(P, InDirection))
        // apply BRDF and add to total
        albedo = S.BRDF(OutDirection, inDirection)
        totalRadiance += InLight * albedo / nSamples
    // return the total
    return totalRadiance
```

More Efficient Integration

```
// MORE EFFICIENT INTEGRATION
// still fails due to infinite recursion
float IndirectLight(Point P, Vector OutDirection)
    // start with no radiance
    totalRadiance = 0
    // and integrate over the incoming hemisphere
    for (n_samples iterations)
        // use Monte Carlo to get vector for incoming direction
        InDirection = MonteCarloHemisphereVector(S.Normal)
        // find the incoming irradiance
        InLight = pathTrace(Ray(P, InDirection))
        // apply BRDF and add to total
        albedo = S.BRDF(OutDirection, inDirection)
        totalRadiance += InLight * albedo / nSamples
    // return the total
    return totalRadiance
```

Inverting the Loops

- We now invert the loops
- Samples are accumulated at the image
- Divide through at the end
- Only really affects indirect light

Image Loop

```
// MODERN RECURSIVE RAYTRACER
// loop through all pixels
for each pixel p_ij
    image[i][j] = 0
// loop through the number of samples we want to take
// or control this with a time-based test
for (n samples iterations
    // loop through all pixels
    for each pixel p_ij
        // add the radiance to the pixel
        // note that Ray's first parameter is the origin of the ray
        // but the destination of the light
        image[i][j] += pathTrace(Ray(Eye, p_ij - Eye)) / nSamples
```

Modified Indirect Lighting

```
// MODIFIED INDIRECT LIGHT
// still fails due to infinite recursion
float IndirectLight(Point P, Vector OutDirection)
    // use Monte Carlo to get vector for incoming direction
    InDirection = MonteCarloHemisphereVector(S.Normal)
    // find the incoming irradiance
    InLight = pathTrace(Ray(P, InDirection))
    // apply BRDF and compute reflection
    albedo = S.BRDF(OutDirection, inDirection)
    // notice that summation is now external to the function
    return InLight * albedo
```


UNIVERSITY OF LEEDS

Incoming vs. Outgoing

- Incoming light:
  - $L_I = \int_{\omega_i \in \boldsymbol{S}_+^2(P)} L(P, -\omega_i) f_r \, \mathrm{d}\omega_i$
- Outgoing light:
  - $L_O = \int_{\omega_o} \int_{\omega_i \in \boldsymbol{S}_+^2(P)} f_r(P, \omega_i, \omega_0)\left(\omega_i \cdot \overrightarrow{n_p}\right) \mathrm{d}\omega_i \, d\omega_o$
- Outgoing light is less (energy conservation)
  - $L_O < L_I$
- And we can include this in the integral

UNIVERSITY OF LEEDS

Extinction Coefficient

- The %age of light that is lost at each bounce
- Represents photons that were *extinguished*:
  - $c_x(P) = \frac{L_I - L_O}{L_I}$
- Used as probability that no bounce happened
  - i.e. terminates the recursion
- Statistically guaranteed to do so eventually
  - so our code will work

Extinction Code

```
// EVIL CHEATY HACK
// terminates probabilistically
float IndirectLight(Point P, Vector OutDirection)
    // test for extinction
    if (RandomRange(0,1) < S.Extinction)
        return 0;
    // use Monte Carlo to get vector for incoming direction
    InDirection = MonteCarloHemisphereVector(S.Normal)
    // find the incoming irradiance
    InLight = pathTrace(Ray(P, InDirection))
    // apply BRDF and compute reflection
    albedo = S.BRDF(OutDirection, inDirection)
    // summation is still external to the function
    return InLight * albedo
```

Recursion

- We invoke pathTrace recursively
- Each bounce generates more rays
  - But eventually all rays terminate
  - Due to extinction coefficient (the base case)
- However, this is why the large run times
- Real-time rendering is about optimising this

Simple Optimisation

- At every bounce, light attenuates
- I.e. less of it carries forward
- So pass that percentage into the recursion
- Terminate when it drops below a threshold
  - e.g. 0.0001

Albedo Termination

```
// path tracing routine
float pathTrace(Ray R, float combinedAlbedo)
    // early termination based on albedo
    if (combinedAlbedo < albedoThreshold)
        return 0
    // total radiance to return
    totalRadiance = 0
    // compute intersection point
    Triangle T = ClosestTriangleAlong(R)
    Surfel S = Intersection(R, T)
    // estimate lighting
    totalRadiance += S.Emission
    // now loop through light sources
    for (each Light)
        // add direct reflection for that light source
        totalRadiance += DirectLight(S, -R.Direction, Light)
    // add indirect light
    totalRadiance += IndirectLight(S, -R.Direction, combinedAlbedo)
    // return the result
    return totalRadiance
```

Albedo Termination

```
// ALBEDO-BASED TERMINATION
// still terminates probabilistically
float IndirectLight(Point P, Vector OutDirection, float combinedAlbedo)
    // test for extinction
    if (RandomRange(0,1) < S.Extinction)
        return 0;
    // use Monte Carlo to get vector for incoming direction
    InDirection = MonteCarloHemisphereVector(S.Normal)
    // apply BRDF and compute reflection
    albedo = S.BRDF(OutDirection, inDirection)
    // find the incoming irradiance
    InLight = pathTrace(Ray(P, InDirection), combinedAlbedo * albedo)
    // summation is still external to the function
    return InLight * albedo
```

Area Luminaires

- Each point on the surface of the light emits
- We want to integrate over this area
  - Even if it has a funny shape
- Solution: Monte Carlo simulation
  - Buried in Luminaire::Position
- The sampling takes care of the integration

UNIVERSITY OF LEEDS

Impulse Reflection

- We want 60% impulse reflection, 40% other
- Treat like the extinction coefficient
- Random number chooses codepath
  - Perfect reflection
    - Uses vector $\overrightarrow{\omega_i} = \vec{n} + (\vec{n} - \overrightarrow{\omega_o})$
  - Standard indirect reflection

Impulse Reflection Code

```
// ADDING IMPULSE REFLECTION
float IndirectLight(Point P, Vector OutDirection, float combinedAlbedo)
    // test for extinction
    if (RandomRange(0,1) < S.Extinction)
        return 0;
    // test for an impulse bounce
    if (RandomRange(0,0) < S.Impulse)
        // for an impulse bounce, take the perfect reflection
        InDirection = 2.0 * S.Normal - OutDirection
        // and set albedo directly
        albedo = S.ImpulseAlbedo
    else
        // use Monte Carlo to get vector for incoming direction
        InDirection = MonteCarloHemisphereVector(S.Normal)
        // apply BRDF and compute reflection
        albedo = S.BRDF(OutDirection, inDirection)
    // find the incoming irradiance
    InLight = pathTrace(Ray(P, InDirection), combinedAlbedo * albedo)
    // summation is still external to the function
    return InLight * albedo
```

Adding Transmission

- Substitute a scattering equation
- $L^{r,out}(P, \omega_o) = \int_{\omega_i \in S^2(P)} L^{in}(P, -\omega_i) f_s(P, \omega_i, \omega_0) \left|\omega_i \cdot \overrightarrow{n_p}\right| \mathrm{d}\omega_i$
- Integrates over *all* incoming directions
- BRDF becomes BSDF
- dot product now has absolute value
- and we add in, out annotations

Sensor Modelling

- Measuring the light is also an integral
  - $m_{ij} = \int_{U \times S^2} M_{ij}(P, \omega) L_{in}(P, -\omega) \left| \omega \cdot \overrightarrow{n_p} \right| dP d\omega$
  - M_{ij} describes the pixel & it's cone of rays
    - Typically the same form for all pixels
    - Also known as the importance function
- Monte Carlo sample points on the pixel
  - They will integrate slightly different rays
  - Embed in Eye::Position or Pixel::Position

Images by

- S7 Martin Lostak
- S24 Tasos Mansour
- S30 Giorgio Trovato
- S31 Pat Whelen