Advanced Computer Graphics Proseminar

Univ.-Prof. Dr. Matthias Harders

Winter semester 2015



Path Tracing

- Target: solving rendering equation

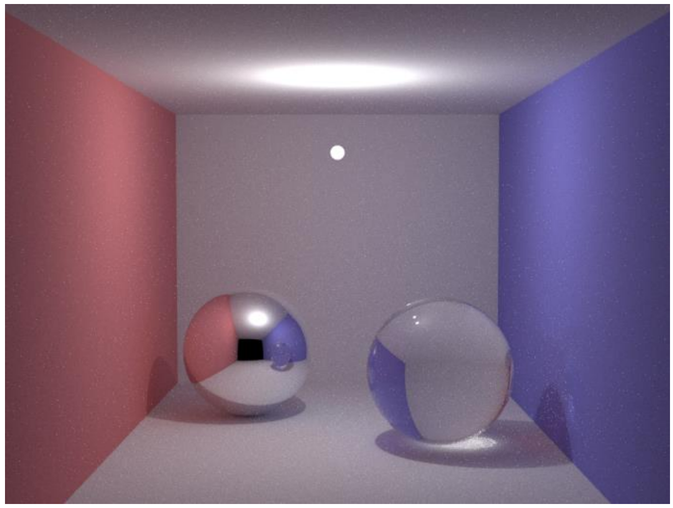
$$L(x \to \omega_o) = L_e(x \to \omega_o) + \int_{\Omega} f(x, \omega_i, \omega_o) \cdot L(x \leftarrow \omega_i) \cos\theta_i \, d\omega_i$$

- Numerical solution with Monte-Carlo integration

$$L(x \to \omega_o) = L_e(x \to \omega_o) + L_r(x \to \omega_o)$$

$$L_r(x \to \omega_o) \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(x, \omega_i, \omega_o) \cdot L(x \leftarrow \omega_i) \cos\theta_i}{p(\omega_i)}$$

Example Code Rendering



Example Code – Main Features

- Renders Cornell box-type scene
- Geometries defined by spheres
- Surfaces perfectly diffuse, specular or transparent
- Explicit sampling of light sources for direct illumination
- Russian Roulette for ray termination
- Computes output image in PPM format

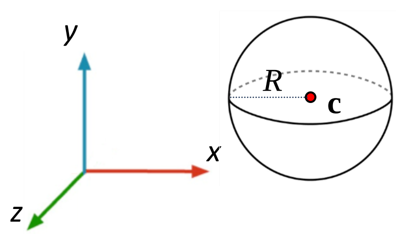
Key Functions

- Intersection of rays with geometry: `Intersect()`
- Calculate radiance recursively: `Radiance()`
- Setup camera, create and send primary rays: `main()`

Ray-Sphere Intersection

- Sphere

$$(\mathbf{x}-\mathbf{c})^2 - R^2 = 0$$



- Intersection test

$$((\mathbf{p}+t\cdot\mathbf{d})-\mathbf{c})^2 - R^2 = 0$$

$$\mathbf{d}^2 t^2 + (2(\mathbf{p}-\mathbf{c})\mathbf{d})t + \left((\mathbf{p}-\mathbf{c})^2 - R^2\right) = 0$$

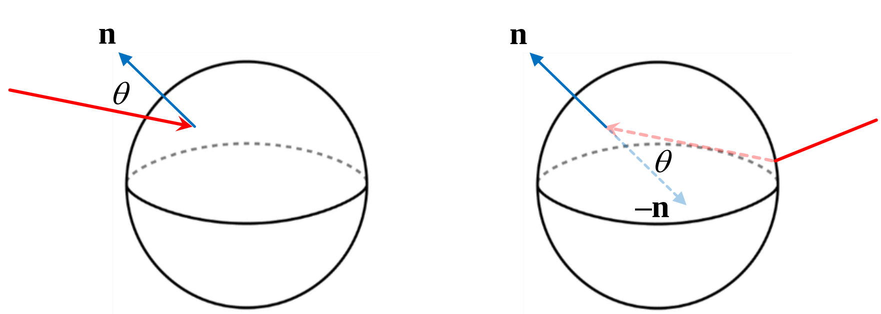
$$\Rightarrow t = b \pm \sqrt{b^2 - \left((\mathbf{op})^2 + R^2\right)}$$

Ray-Sphere Intersection

```
Vector op = position - ray.org;
double eps = 1e-4;
double b = op.Dot(ray.dir);
double radicant = b*b - op.Dot(op) + radius*radius;

if (radicant < 0.0)  return 0.0;
else  radicant = sqrt(radicant);

double t = b - radicant;
if(t > eps)  return t;

t = b + radicant;
if(t > eps)  return t;

return 0.0;
```


Determining Intersection Normal

- For transparent objects intersection may be from inside, requiring flipped normal for computations



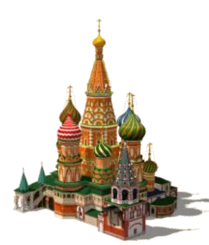
Determining Intersection Normal

```
Vector hitpoint = ray.org + ray.dir * t;
Vector normal = (hitpoint - obj.position).Normalized();
Vector nl = normal;

if (normal.Dot(ray.dir) >= 0)
    nl = nl*-1.0;
```


Russian Roulette

- Terminate recursion stochastically
- Assume probability of termination $q \in [0,1]$
- Re-weighting required to remain unbiased
- Typically based on surface reflectivity (e.g. maximum RGB value)
- Employ new, scaled, unbiased estimator

$$\hat{I}_{RR} = \begin{cases} \frac{1}{q}\hat{I} & \xi < q \\ 0 & \xi \geq q \end{cases}$$



Russian Roulette

```
Color col = obj.color;
double p = col.Max();

if (depth > 5 || !p)
{
    if (drand48() < p)
        col = col * (1/p);
    else
        return obj.emission * E;
}
```


Lambertian Diffuse Reflection

- Diffuse reflection split into different components
  - Indirect illumination from other surfaces
  - Direct illumination from light sources
  - Direct emission (for points on light source)
- Note: local orthogonal coordinate system ($\mathbf{u}$,$\mathbf{v}$,$\mathbf{w}$) on surface at intersection point

Non-Uniform Hemisphere Sampling

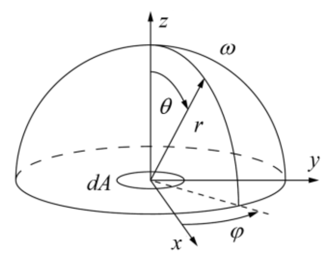
- For Monte-Carlo integration of indirect illumination, generate random reflection vector
- Use non-uniform probability density function, proportional to cosine weighted solid angle

$$p(\omega) = \frac{\cos\theta}{\pi}$$

$$x = \cos(2\pi \cdot \xi_0) \cdot \sqrt{\xi_1}$$

$$y = \sin(2\pi \cdot \xi_0) \cdot \sqrt{\xi_1}$$

$$z = \sqrt{1 - \xi_1}$$

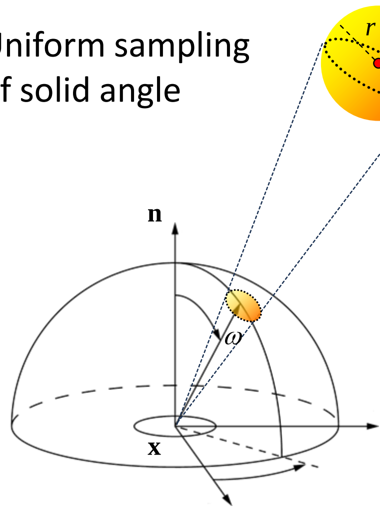


Non-Uniform Hemisphere Sampling

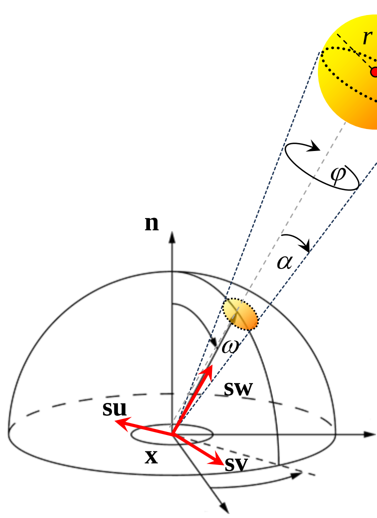
```
double r1 = 2.0 * M_PI * drand48();
double r2 = drand48();
double r2s = sqrt(r2);

…

Vector d = (u * cos(r1) * r2s +
            v * sin(r1) * r2s +
            w * sqrt(1 - r2)).Normalized();
```

Uniformly Sampling Spherical Light Source

Uniform sampling
of solid angle



Uniformly Sampling Spherical Light Source

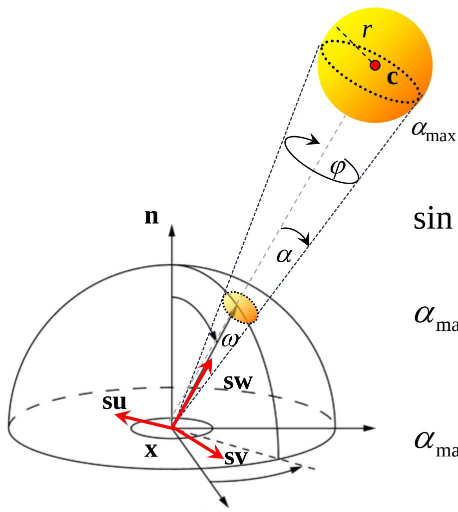


$$\mathbf{sw} = \frac{\mathbf{c} - \mathbf{x}}{\|\mathbf{c} - \mathbf{x}\|}$$

$$\mathbf{sv} = \frac{\mathbf{sw} \times \mathbf{n}}{\|\mathbf{sw} \times \mathbf{n}\|}$$

$$\mathbf{su} = \frac{\mathbf{sv} \times \mathbf{sw}}{\|\mathbf{sv} \times \mathbf{sw}\|}$$

Uniformly Sampling Spherical Light Source



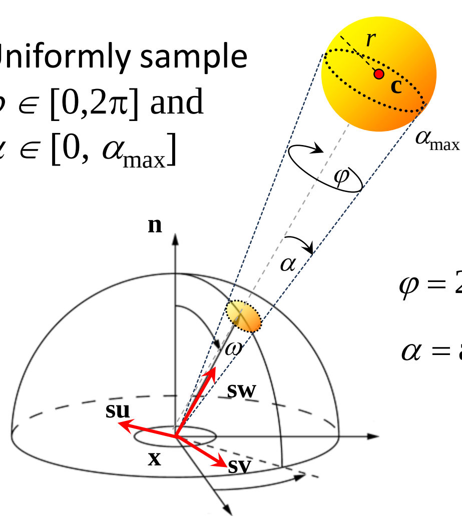
$$\sin\alpha_{max} = \frac{r}{\|\mathbf{x}-\mathbf{c}\|}$$

$$\alpha_{max} = \arcsin\frac{r}{\|\mathbf{x}-\mathbf{c}\|}$$

$$\alpha_{max} = \arccos\sqrt{1-\left(\frac{r}{\|\mathbf{x}-\mathbf{c}\|}\right)^2}$$

Uniformly Sampling Spherical Light Source

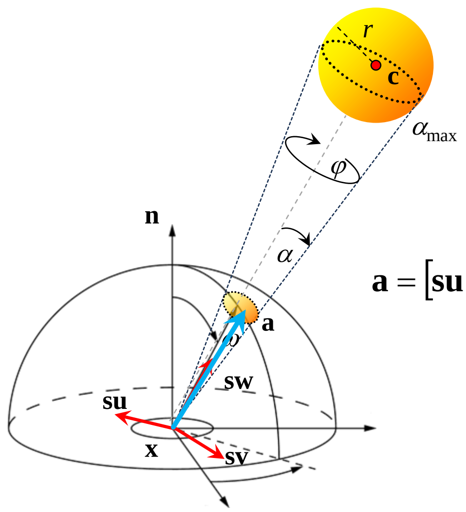
Uniformly sample
$\varphi \in [0,2\pi]$ and
$\alpha \in [0, \alpha_{max}]$



$$\varphi = 2\pi \cdot \xi_1$$

$$\alpha = \arccos(1+\xi_2(\cos\alpha_{max}-1))$$

Uniformly Sampling Spherical Light Source



$$\mathbf{a} = [\mathbf{su} \quad \mathbf{sv} \quad \mathbf{sw}] \cdot \begin{bmatrix} \cos\varphi \sin\alpha \\ \sin\varphi \sin\alpha \\ \cos\alpha \end{bmatrix}$$

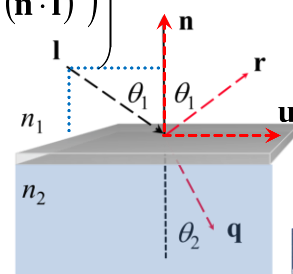
Uniformly Sampling Spherical Light Source

```
Vector diff = hitpoint - sphere.position;
double cos_a_max =
      sqrt(1.0 - sphere.radius * sphere.radius /
      (diff).Dot(diff));
double eps1 = drand48();
double eps2 = drand48();
double cos_a = 1.0 - eps1 + eps1 * cos_a_max;
double sin_a = sqrt(1.0 - cos_a * cos_a);
double phi = 2.0*M_PI * eps2;
Vector l = su * cos(phi) * sin_a +
           sv * sin(phi) * sin_a +
           sw * cos_a;
l = l.Normalized();
```

Determining Refraction Vector

- Task: given light and normal vectors, as well as indices of refraction, determine refraction vector $\mathbf{q}$
- Derive using local coordinate system given by basis vectors $\mathbf{n}$ and $\mathbf{u}$

$$\mathbf{q} = \frac{n_1}{n_2}\mathbf{l} - \mathbf{n}\left(\mathbf{n}\cdot\mathbf{l}\frac{n_1}{n_2} + \sqrt{1-\left(\frac{n_1}{n_2}\right)^2\left(1-(\mathbf{n}\cdot\mathbf{l})^2\right)} \right)$$



Determining Refraction Vector

```
double nc = 1;
double nt = 1.5;
double nnt = nc/nt;
…
double ddn = ray.dir.Dot(nl);
double cos2t = 1 - nnt * nnt * (1 - ddn*ddn);
if (cos2t < 0)
    return obj.emission +
    col.MultComponents(Radiance(reflRay, depth, 1));
…
Vector tdir = (ray.dir * nnt - normal *
              (ddn * nnt + sqrt(cos2t))).Normalized();
```

Schlick´s Approximation

- Reflectivity for normal incidence, i.e. $\theta_1 = 0$

$$R_0 = \frac{(n_2 - n_1)^2}{(n_2 + n_1)^2} \qquad T_0 = 1 - R_0$$

- Approximation of reflectivity at incidence other than zero, according to Schlick

$$R = R_0 + (1 - R_0)(1 - \cos\theta_1)^5$$

Schlick´s Approximation

```
double a = nt - nc;
double b = nt + nc;
double R0 = a*a / (b*b);

double c = 1 + ddn;
…
double Re = R0 + (1 - R0) *c*c*c*c*c;
double Tr = 1 - Re;
```

Programming Assignment 2

1) Extend code to include objects represented by triangle meshes (reuse prior development)
2) Simulate a thin lens, and thus depth of field effects
3) Include glossy and translucent materials (i.e. beyond perfect specular reflection/transmission)

Tasks for Next Week

- Today: submit solution for 1st programming assignment in OLAT
- Be prepared for presentation of solution next week
- Start with 2nd programming assignment
- Prepare proposal for final project

Proseminar Schedule

Date	Topic	Remark
12.10.	Introduction	
19.10.	Theory – Radiometry	Radiosity example code
26.10.	*(no proseminar - Nationalfeiertag)*	
2.11.	*(no proseminar - Allerseelen)*	
9.11.	Discussion of Radiosity code	Programming assignment 1
16.11.	Programming support and advice	
23.11.	Theory – Random sampling	Path Tracer example
30.11.	Discussion of Path Tracer code	Programming assignment 2, *Hand-in PA1*
7.12.	Presentation of solutions	
14.12.	Programming support and advice	*Project proposal (21.12. Hand-in PA2)*
	Christmas break	
14.1.	Presentation of solutions	Presentation of solutions
21.1.	Geometric Modelling	
28.1.	Programming support and advice	
4.2.	Project presentation	*Submission final project*