General Representation of Light Sources in Photorealistic Computer Graphics

1 Introduction

The goal of this project was to implement in Java a representation of light sources suitable for rendering based on Monte Carlo solution approaches to the rendering equation.

The rendering equation for monochromatic light can be expressed as

$$L(x,\omega) = L_e(x,\omega) + \int_\Omega \rho(x,\omega',\omega) L(RT(x,-\omega'),\omega') \cos\theta' d\omega' \qquad (1)$$

where the unknown $L(x,\omega)$ is the radiance ("light intensity", measured in $Wsr^{-1}m^{-2}$) leaving the differential area around the surface point x in a differential angle around the direction ω. The given function $L_e(x,\omega)$ is zero everywhere except for points x at the light sources. The given function $\rho(x,\omega',\omega)$ describes the surface materials, more precisely the scattering of light entering the point x from the direction ω'. θ is the angle between the surface normal at the point x and the direction ω. $RT(x,\omega)$ is a ray tracing function which returns the nearest surface point (different from x) from the point x in the direction ω.

The rendering equation is a Fredholm (recursive) integral equation of the second kind. Approaches to solving the equation can be roughly divided into two categories: direct and simplifying. Simplifying approaches such as radiosity and ray tracing transform the rendering equation into a simpler one. However, this not only disallows simulation of all light phenomena governed by the rendering equation but also imposes constraints on the input scene (geometry, materials, light sources, camera). Direct approaches work with unmodified Eq. 1 and unfold the recursion into a series of nested integrals. Although these series are infinite, an iterative evaluation of the integrals converges to the solution of Eq. 1.

2 Rendering Equation and Light Sources

The assumption of the monochromaticity of light can be obliterated by extending the radiance function with the wavelength λ (which ranges in the visible spectrum from ca. 400 nm to ca. 700 nm). The rendering equation

then becomes

$$L(x,\lambda,\omega) = L_e(x,\lambda,\omega) + \int_\Lambda \int_\Omega \rho(x,\lambda',\omega',\lambda,\omega) L(RT(x,-\omega'),\lambda',\omega') \cos\theta' d\omega' d\lambda' \quad (2)$$

In direct solution methods, light sources are generators of (random) beams which transport the radiance from points on the light sources' surfaces. The function L_e serves as a probability distribution function. Sampling of a single light source involves storing two structures in each **Beam**: **SpatialData** (x and ω) and **ColorData** (λ). In order to distribute the total power of the source, the number of generated beams is updated in the light source from which the beams originated. Thus each light source should implement methods `Beam getNextBeam()` which returns a random beam, and `long getNumberOfBeams()` which returns the number of beams which have been generated by this light source. Besides SpatialData and ColorData, each beam stores a reference to its parent light source. Using Java serialization, light sources can be then added to an arbitrary renderer which supports their type.

Therefore, every light source implements the following interface:

```
interface LightSource
extends java.io.Serializable {
        Beam getNextBeam();
        long getNumberOfBeams();
}
```

Each beam implements methods of the following abstract class:

```
public Beam {
        public SpatialData sd;
        public LightSource parent;
        public ColorData c;
}
```

2.1 Representation of color in beams

Human eye has 3 types of cones (S,M,L) which are sensitive to different, overlapping parts of visible light spectrum. The perceived color depends not

only on the wavelengths hitting the eye, but also on the number of photons (beams) which carry the wavelengths.

The **RGB color model** represents color as a triplet $[r, g, b]$ which expresses how much red, green, and blue is measured by the eye's cones. This model is very useful in output devices such as monitors or images stored on a disk. It is also frequently used in ray tracing to represent colors of light sources and scattering of light on surfaces (the Phong model). Although our representation of light sources is general enough to accommodate the RGB model in light sources and beams, we are aware of that an $[r, g, b]$ triple is a very poor approximation of the visible spectrum. It is insufficient for rendering effects such as refraction, fluorescence etc.

2.1.1 Spectral power distribution

Spectral power distribution (SPD) is a function which describes how much energy is radiated on each wavelength in the visible spectrum. (A data format with the same name is used by e.g. manufacturers of light bulbs.) In the context of this work, we use SPD as a probability distribution function and require that it be normalized, i.e. its integral over the whole spectrum be equal to 1. This also simplifies the scaling of the power of a light source with one additional parameter (stored in the light source outside of SPD).

An SPD implements the following interface:

```
interface SpectralPowerDistribution{
        double getNextLambda();
        double getValue(double lambda);
        double[] getFirstAndLastLambda();
}
```

where `double getNextLambda()` generates one random sample λ from the SPD distribution, `double getValue(double lambda)` returns the function value of the SPD, `getFirstAndLastLambda()` returns the limits of the SPD's domain.

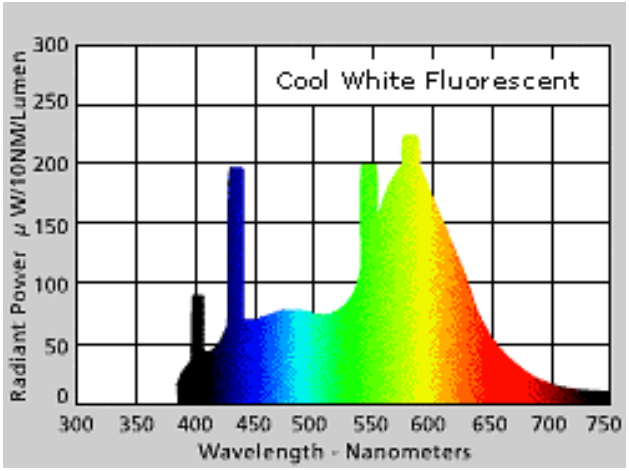

Figure 1: An example of an (unnormalized) SPD, taken from this website

2.2 Sampling of wavelengths and beams

The generation of random beams in SpatialData and random wavelengths in ColorData leads to the problem of discrete samples from a given probability distribution. The number of samples is potentially infinite, but they are generated one by one and their count is kept.

Let us demonstrate the problem on an example.

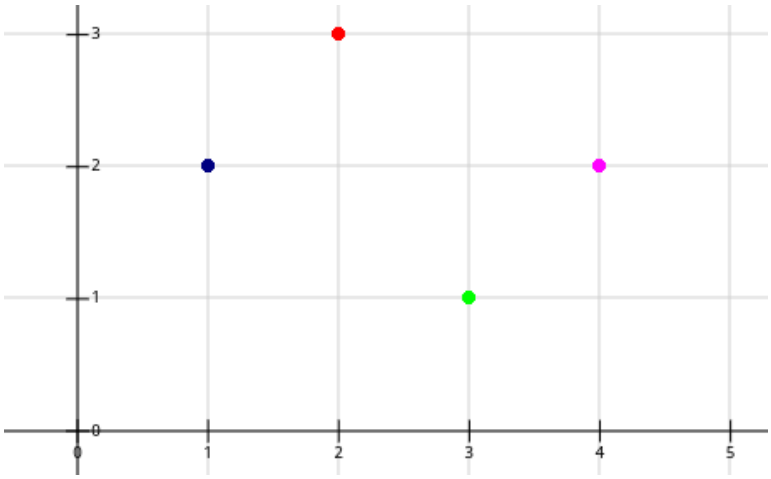

Figure 2: Discrete samples taken from a discrete (not necessarily normalized) probability distribution. X is what we want to get, Y is the weight (frequency of appearance) of the desired X.

In a long run, we want to obtain a sequence with the same amount of 1s and 4s, but only half as many 3s as 1s (or 4s) and three times more 2s than 3s. A simple method to generate such a sequence is to compute the sum I of functional values from 1 to 4 ($I = 8$), then produce a uniform random number $m \in [0, 1)$, and then keep summing the functional values for X from 1 to 4 until their sum S is greater than mI. The number X which we are after is the last (the largest) X for which $S \leq mI$.

The algorithm for sampling an evenly sampled discrete probability distribution is:
Input: A discrete probability distribution function $f(X)$ defined on a uniformly spaced set $\{a, ..., b\}$.
Output: A random sample X from the probability distribution $f(X)$.

1. Set $I = \sum_{n=a}^{b} f(n)$. Note that $I = 1$ for a "genuine" (normalized) probability distribution function $f(X)$.

2. Generate a uniform random number $m \in [0, 1)$.

3. Return the largest X such that $\sum_{n=a}^{X} f(n) \leq mI$.

2.2.1 Linear gap filling functions

The probability distribution functions we deal with are usually not discrete, although their representations are. For example, consider an SPD which is defined on an interval of visible wavelengths, but is represented by uniformly spaced discrete points in which the function values are known (measured by a spectrometer). When we treat such an SPD as a discrete function, then the distribution of samples generated by the sampling algorithm above will be "choppy". It is desirable to interpolate the function values between the discrete points. The interpolation can be perceived as an extension of the discrete function $f(X)$ with"gap filling" functions $g_n(X)$, where the function $g_n(X)$ is defined on the interval between the successive n-th and $(n+1)$-th points of $f(X)$.

Let a discrete f(X) be defined at points $x_1, \ldots, x_N$. A linear gap filling function $g_n(x)$ is then defined on the interval $[x_k, x_{k+1}]$ as

$$g(x) = \frac{f(x_{k+1}) - f(x_k)}{x_{k+1} - x_k} * (x - x_k) + f(x_k) \tag{3}$$

The following sampling algorithm returns a real random r according to a gap-filled (not necessarily normalized) probability distribution function $f(X)$:

1. Compute $I = \sum_{i=1}^{n} \int_{x_i}^{x_{i+1}} g_i(x)dx$.
2. Generate a uniform random number $m \in [0, 1)$.
3. Find the largest j such that $\sum_{i=1}^{j} \int_{x_i}^{x_{i+1}} g_i(x)dx \leq mI$.
4. Return r from the interval $[x_j, \ldots, x_N]$ such that $(\sum_{i=1}^{j} \int_{x_i}^{x_{i+1}} g_i(x)dx) + \int_{x_{j+1}}^{r} g_{j+1}(x)dx = mI$.

3 Examples of concrete light sources

3.1 Conventions

Throughout this section, the following mathematical and physical conventions are used:

- We work with 3-dimensional Euclidean space. However, points, directions and normals are represented by 4-valued vectors $[x, y, z, w]$, whereby the last value w is implicitly 1 unless stated otherwise.
- Right-handed cordinate system is used, i.e. x is the horizontal direction (increasing from left to right), y is the vertical direction (increasing from bottom to top), z is the depth (increasing inwards).
- Transformation matrices are represented by matrices 4x4 row major order.
- Positive rotations, ordering of vertices (e.g. of a triangle) etc. are counter-clockwise.
- Color is represented by SPD. Wavelengths are given in nanometers, not meters.

A beam of light (a photon carrying energy from its parent light source on a wavelength λ) can thus be represented as

```
public class Beam {
        //SpatialData
        public Vector3 origin, direction;
        //LS parent
        public LightSource parent;
        //ColorData
        public double lambda, scale = 1, level = 1;
}
```

Note that instead of a using Vector, we could store direction as 2 angles. This would save some memory but would increase the time complexity of computations.

The variable `Beam.scale` represents the weight of a beam, hence the power of the beam is equal to $\frac{Beam.parent.getPower()}{Beam.parent.getNumberOfBeams()} * Beam.scale$.

The class `spdLightSource` is still an abstract class, but it is more concrete than `LightSource` in that it uses SPD to represent the color. The variable `power` is used to scale the intensity of the SPD.

```
public abstract class spdLightSource
implements LightSource {
        private double power;
```

```
        private SpectralPowerDistribution spd;
        private long number_of_beams = 0;

        public spdLightSource(
                SpectralPowerDistribution _spd,
                double _power)
        {spd = _spd; power = _power;}

        public double getPower()
        {return power;}
}
```

The variable `number_of_beams` is increased every time `getNextBeam()` is called. The variable `power` is the total energy released by this light source, hence the energy on a wavelength `lambda` is equal to `power*spd.getValue(lambda)`.

We are now ready to show the implementation of various concrete light sources. They all are based on the abstract class `spdLightSource`.

3.2 Laser

A simple example of a concrete light source is `Laser` which shines from one point in one direction (and often emits light within a very narrow spectrum):

```
public class Laser extends spdLightSource{
        private Vector3 position, direction;

        public Laser(SpectralPowerDistribution _spd,
                double _power, Vector3 _position,
                Vector3 _direction)
        {
                super(_spd, _power);
                position = _position;
                direction = _direction;
        }

        public Beam getNextBeam()
        {
                Beam b = new Beam();
```

```
                b.lambda = spd.getNextLambda();
                b.origin = position;
                b.direction = direction;
                number_of_beams++;
                return b;
        }
}
```

3.3 Sky

The lights source `Sky` is perceived as an infinitely distant rectangular light source covered with `Laser`s radiating light in parallel beams. It resembles the Sun illuminating the Earth (as the Sun is very far, the beams hitting the Earth are almost parallel).

```
public class Sky extends spdLightSource{
        private Vector3 A, B, direction,
                fromAverticalDir,
                fromAhorizontalDir;
                // A and B are the corners
                // of the sky rectangle
        private Laser laser;
        private Random verticalr, horizontalr;

        public Sky(SpectralPowerDistribution _spd,
                double _power, Vector3 _A, Vector3 _B,
                Vector3 _direction)
        {
                ... initialization ...
        }

        public Beam getNextBeam()
        {
                Beam b = laser.getNextBeam();
                b.origin +=
                verticalr.getDouble() *
                fromAverticalDir;
```

```
                b.origin +=
                horizontalr.getDouble() *
                fromAhorizontalDir;
                number_of_beams++;
                return b;
        }
}
```

3.4 Fading Spot Light

All beams of `FadingSpotLight` originate from the same point and are distributed within a cone extending from that point (they are therefore not parallel as in `Laser` and `Sky`).

Until now, described light sources would shine uniformly everywhere within some boundaries. This is usually not the case, because nearly all (except lasers an such) lights emit beams witch are not parallel. Illumination from light sources is naturally dimmer, the greater is the distance form Beam origin to intersection point. So if you shine Uniform Spot Light perpendicularly to flat surface, the light would be dimmer the further away you are form center.

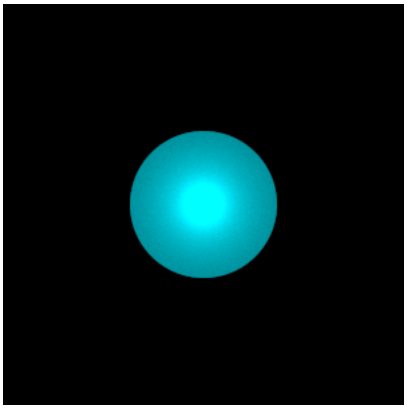

Figure 3: Simple Spot Light - light is emitted from 1 point uniformly in cone

Fading Spot Light is like Uniform Spot Light, but it is deliberately dimmer further from center of cone.

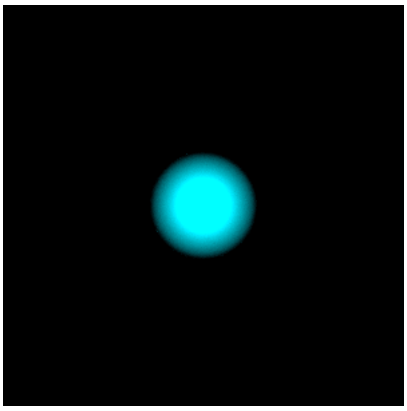

Figure 4: Fading Spot Light with slightly higher power

To do that, we can create sample set and turn it it into weighted random distribution. Spot light needs (besides position and direction) another parameter - cone angle, which is angle from direction in which beams can be generated. Fading spot light also needs parameter that describes fading. In this example, that parameter is `fade_per_angle`, which describes how much percent dimmer should light shine. From `cone_angle` and `fade_per_adngle`, we can calculate `fade_angle` as (`cone_angle - 1/fade_per_angle`), which is an angle from which onward, light intensity is lowered by value of `fade_per_angle`, per angle. Our Sample set would have $f(x) = 1$ for x from [0,`fade_angle`) and $f(x) = 1 - (\texttt{cone\_angle} - x) * \texttt{fade\_per\_angle}$ for other. Fading spot light class can look like this:

```
public class  FadingSpotLight extends spdLightSource(){
  private Vector3 perpendicular;
  private WeightedRandomGenerator wrg;
  private Random circlerot;
        . . .

  public FadingSpotLight( . . .){
                . . .
        perpendicular = direction.getRandomCross();
        wrg = new WeightedRandomGenerator(. . .);
  }
. . .
  public Beam getNextBeam(){
```

```
            Beam b = new Bean();
            b.lambda = spd.getNextLambda();
            b.origin = position;
            double angle = wrg.nextDouble();
            b.direction = direction.rotateTowards(
              perpendicular, toRadians(angle));
            angle = circlerot.nextDouble()*360;
            b.direction = b.direction.rotateAround(
              direction, toRadians(angle));
            number_of_beams++;
            return b;
    }
  . . .
}
```

`WeightedRandomGenerator wrg` is our weighted random distribution for light intensity which is used to generate angle from [0, `cone_angle`), with increasing bias against bigger angles. Spot light is simetrical on circles(around direction vector), so we can uniformly randomly turn new Beam.direction around spot light direction. This way, fading spot light has adjustable fade towards light cone edge.