General representation of light sources in photorealistic computer graphics

1 Introduction

This project models various light sources and describes their generalization for rendering based on rendering equation using Monte Carlo method. The rendering equation can be expressed as
$I(x,x') = g(x,x')[e(x,x') + \int_S p(x,x',x'')I(x',x'')dx]$, where:
$I(x,x')$ is related to the intensity of light passing from point x' to point x
$g(x,x')$ is "geometry" term
$e(x,x')$ is related to the intensity of emitted light from x' to x
$p(x,x',x'')$ is related to the intensity of light scattered from x'' to x by a patch of surface at x'

2 Light Source

Direct approach to solve The rendering equation require that light sources act as "light intensity in direction generators". Sampling of such light source will result in **Beams** - structures containing **SpatialData** or x and x' and **ColorData** or intensity of emitted light (with some color - wavelength), what is special case of $I(x,x')$.Because sampling can be done indefinitely, ColorData cannot contain definitive value. That has to be interpolated using data from all other Beams and intensity of light source. Light source has to emit all of its power, which means that sum of all beams intensities is equal to the intensity of Light source they originate from. Knowing this, we can set all Beam intensity to (SourcePower)/(NumberOfBeams) and rely on extensive enough sampling of Light source, which will generate different colors with different weights. Thus every Light source should implement methods `Beam getNextBeam()` - which returns some Beam, and `long getNumberOfBeams()` - which returns how many Beams were created by this light Source. Beams have to contain SpatialData, ColorData and reference for their parent light source. Light Sources can be then add to renderer which supports their Beam type by serialization.

Therefore, every Light Sources must implement this interface:

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

Beam is abstract class that generally looks like this:

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

2.1 Beam color representation

Human eyes have 3 cones (S,M,L) which perceive color. They are sensitive to different, overlapping parts of visible light spectrum, and are more sensitive to different wavelengths in their corresponding part of light spectrum. This means that color does not depend only on which wavelengths are hitting your eye, but also on how many fotons with same/different wave length are there.

Based on this is **RGB color model**, which represent color as RGB triplet (r,g,b) - how much of each of the red, green, and blue is included. Because of that, its impossible to reconstruct full spectrum of light from RGB, which is needed for realistic representation of light effects like refraction of light, fluorescence, color perception of objects illuminated with differently colored lights (Apparent color), and probably more. Instead of RGB, I will use in this text Spectral representation of color, using Spectral Power Distribution so aforementioned effects would be more realistic.

2.1.1 Spectral power distribution

Spectral power distribution or SPD for short, represents how much energy is radiated on each wavelength. It can be directly used as wavelength weight in weighted random generation. **SPD should have integral equal to 1**, so its easy to scale their power on Light source.

All SPDs have to implement this interface:

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

where `double getNextLambda()` is for wavelength sampling and `double getValue(double lambda`
returns value of lambda on normalized SPD.

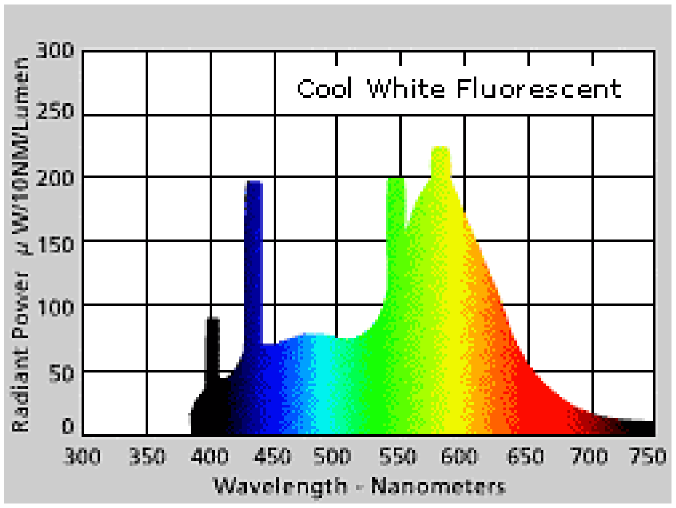

Figure 1: SPD example from this website

2.2 Sampling

This sections describes how to take discrete samples and turn them into weighted random generators. Such process is needed for Beams SpatialData and ColorData generation.

Discrete samples - finite number of outputs with their counts.

Weighted random generator - return 1 output between lowest discreet sample and highest. After finite calls to this generator, resulting output distribution would be similar to the original sample distribution.

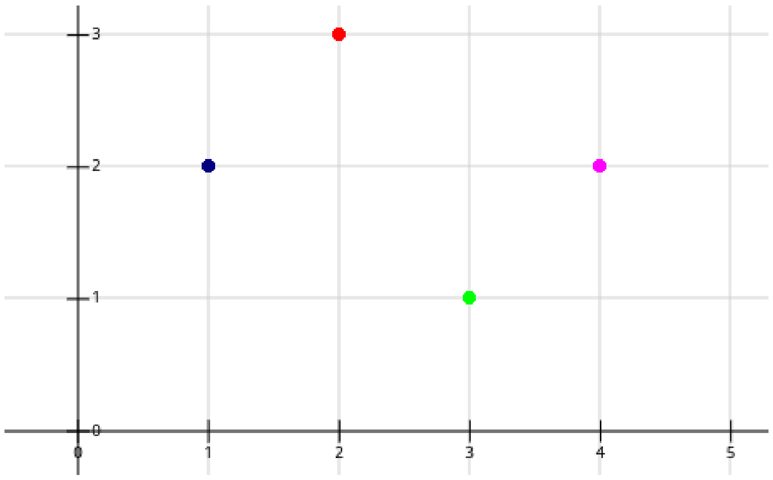

Figure 2: discrete samples

X is what we want to get, Y is the weight of desired X.

That means that we want same amount of 1s as 4s, but only half as many 3s as 1s (or 4s) and three times more 2s than 3s. An easy solution is to compute the sum of functional values form 1 to 4 of this sample, which is 8, and then produce random number m form [0,1) , multiply it with found sum and then sum functional values until our sum is greater than m*8. Our random number from this weighted distribution is the last X used in sum. *I use sum instead of integral, because we do not care about step size in this case.*

Generalization: We have samples on uniformly spaced set {a,...,b} with values of f(x), x belongs to {a,...,b}

1. Calculate $I = \sum_{n=a}^{b} f(n)$
2. Calculate $i = m * I$, m is random number from $[0, 1)$
3. Find greatest r from $\sum_{n=a}^{r} f(n) < i$, where r is result

Problem with this solution is that it creates "Choppy" distribution,what can be suppressed by adding more samples (what would reduce "choppynes", but increase number of iterations), but not in all cases, or by replacing gaps with some function that is easy to integrate which would remove "choppynes" completely and make this generator continuous, in which case we have to solve $\int_a^r h(x) = m * \int_a^b g(x)$, where r is the result and $h(x)$ is new continuous function composed from $f(x)$ and continuous filler functions $g_n(x)$.

2.2.1 Linear function as gap filler

To find our function $g_n(x)$ for gap filling between A and B we will use this formula: $y - y1 = m * (x - x1)$. In our case, $m = (f(B)–f(A))/(B - A)$, $y1 = f(A)$ and $x1 = A$, $y = \frac{f(B)-f(A)}{B-A} * (x - A) + f(A)$.

Let $c_1 = \frac{f(B)-f(A)}{B-A}$ and $c_2 = \frac{f(B)*A+f(A)*B}{B-A}$, then$g_n(x) = c_1 * x - c_2$, x belongs to$[A, B)$ and $\int g_n(x) = c_1 \int xdx - c_2 \int 1dx = \frac{c_1}{2}x^2 - c_2 x = x(\frac{c_1}{2}x - c_2)$. Now we can proceed similarly as before:

1. Calculate I as sum of integrals for $g_n(x)$ on their respective $[A, B)$ interval.
2. Calculate $i = m * I$, m is random number from $[0, 1)$
3. Find greatest r_1 from $\{a, .., b\}$ that sum of integrals for $g_n(x)$ on their respective $[A, B)$ is $< i$
4. Solve $\int_{r_1}^{r_2} g_n(x) = i - r_1 * I$
5. result is $r = r_1 + r_2$

In $\int_{r_1}^{r_2} g_n(x) = i - r_1 * I$, the $r_2 = A$ for that $g_n(x)$, so : $r_2(\frac{c_1}{2}r_2 - c_2) - A(\frac{c_1}{2}A - c_2) = i - r_1 * I$ what is quadratic equation. $D = c_2^2 - 4\frac{c_1}{2} * [A(\frac{c_1}{2}A - c_2) - i + r_1 * I]$, so $r_2 = \frac{-c_2 +- \sqrt{(D)}}{c_1}$.

2.2.2 Same amount of wavelengths, but different scales

Previously mentioned approach applied in Light source generates different amounts of Beam with different wavelengths (colors).

Complementary approach would be generating same amounts of all wavelengths(colors), but adding scale/weight value to beams. Scale can be value of lambda on normalized SPD (integral = 1). Power of one Beam would be harder to express.

2.3 Conventions

This document follows these math, physics, vector space and rendering conventions:

- Using Vectors [x,y,z,w] (4th value is implicitly 1) for points, directions and normals

- 4x4 row major order matrices for transformations.
- Counter clockwise vertices order
- Right-handed system - x is horizontal (bottom of monitor), y is vertical, z is depth (exiting from monitor towards viewer is positive value, entering into monitor is negative value)
- wavelength is in nanometers, not meters

2.4 Examples

As mentioned before, i use SPD for color and three-dimensional Euclidean space as space representation.

Thus my Beam can look like this:

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

Instead of direction as Vector, we can store it as 2 angles, to save memory, but it would take more time to use them in computations. `Beam.scale` is beam weight, so beam power is equal to $\frac{Beam.parent.getPower()}{Beam.parent.getNumberOfBeams()} * Beam.scale$. This and `level` variable will be described further in Renderer section. They are used when parent light source power is not known (when using less wasteful path creation in bidirectional rendering) and are not needed for simple rendering from Light source.

I will also define an abstract class spdLightSource, which takes SPD and power as parameters - for easy color and intensity changing. All examples are based on this class.

```
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;}
}
```

Variable `number_of_beams` is increased every time `getNextBeam()` is called, `power` is all the energy released by this light source, so energy on wavelength `l` is equal to `power*spd.getValue(l)`.

2.4.1 Laser

Probably the simplest example of light source is Laser. It shines from one point in one direction and often emits light with a very narrow spectrum - only few wavelengths.

Its implementation can look like this:

```
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;
        }
}
```

2.4.2 Sky

Sky is based on how light beams from sun appears on earth. Because sun is very far away, in everyday human life they appear to be parallel. So sky is like array of laser. Thus its implementation can look like this:

```
public class Sky extends spdLightSource{
        private Vector3 A,B,direction , fromAverticalDir ,
                        fromAhorizontalDir;
        private Laser laser; // position is A
        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;
        }
}
```

Sky generates Beams using Laser and then changes Beams origin coordinates to be within rectangle with corners A and B, so Sky effectively illuminates only as big space as is its rectangle.

2.4.3 Fading Spot Light

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.

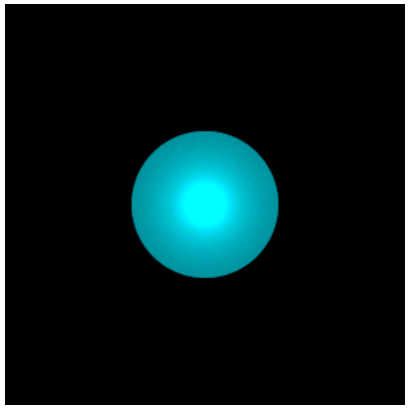

Figure 3: Simple Spot Light - light is emitted from 1 point uniformly in cone

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

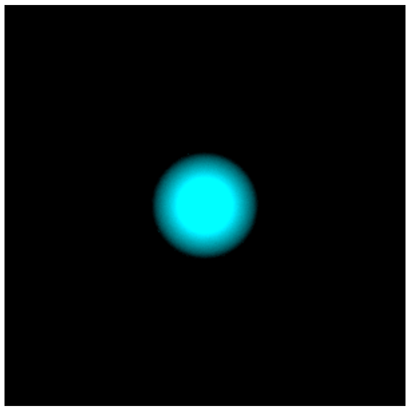

Figure 4: Fading Spot Light with slightly higher power

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 $f(x) = 1 - ($`cone_angle` $-x)*$`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.

2.4.4 Simple "from-LightSource" rendering

Process of simple rendering using Diffuse material, and/or forced reflection/refraction and only 1 light source can be summarized as following:

1. Set up scene objects (Geometry and simple "material functions"), add Light source, add simple deterministic camera.
2. In cycle, call `getNextBeam()`, check if it intersects with diffuse object, if yes: send that beam to camera from intersection point and generate new Beam. If no, and it intersect with nothing, generate new Beam, if it intersect with other than diffuse object, call its material function which will return new Beam. Check if this beam intersects with some scene object. Repeat till iteration limit per beam is reached. If this limit is reached, generate new beam.
3. After some iterations, let camera generate picture.

Because of lacking material functions, this does not provide photorealism, but it is good enougth for some test picture.

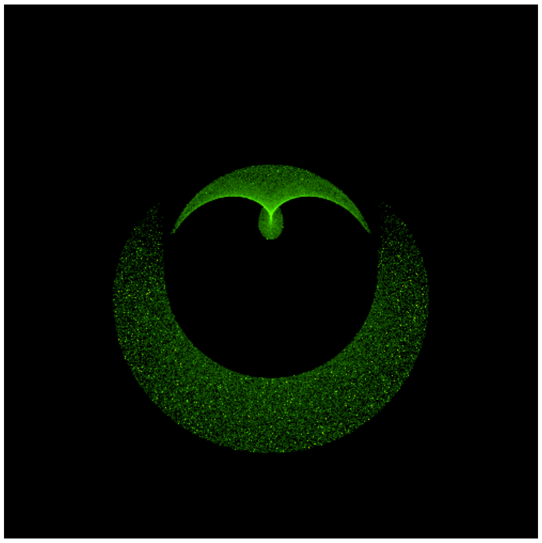

Figure 5: Caustic only, on ring, low beam count

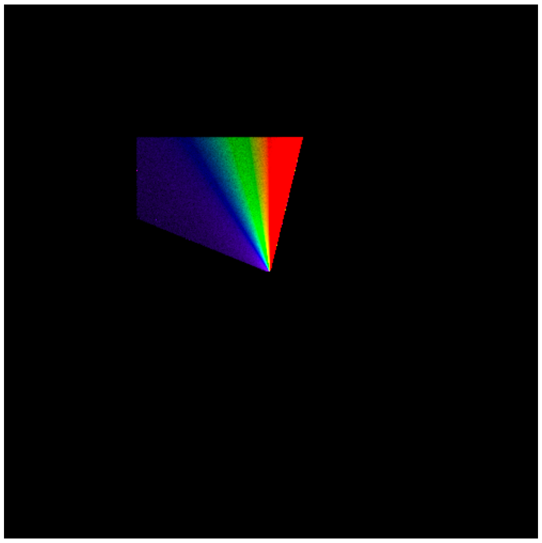

Figure 6: Top down, refraction of white light on prism

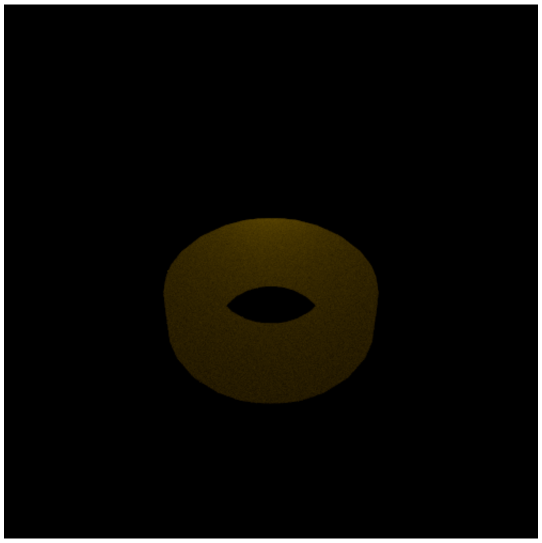

Figure 7: Diffuse torus, illuminated by SimpleSpotLight

3 Renderer

Next step is to create rendered using bidirectional path tracing, with SPD and Power as color representation which enables easy use of known light interaction rules and thus should result in easily created photo-like images.

Following sections are stubs, they are subject to future development of this project

3.1 Scene

Contains SceneObjects, Cameras (CameraManager) and lights (LightSource-Manager), manages light/camera path tracing.

3.2 Scene Objects

Something that can interact with beams (light/camera). Provides ray intersection function, which returns new beam/LightSource using relevant material function. Scene should contain SceneObjectManager containing all SO.

3.3 Camera

Can create image from detected beam data, Can create camera beams, is aware of "master light source" - LightSourceManager Scene should contain CameraManager, CameraManager contains all other sensors (Cameras)

3.4 Light Sources

3.5 Materials

Have material function which takes beam and returns new Beam/LS.