Report - zimný semester
Ročníkový projekt
Renderovacia rovnica a všeobecná reprezentácia kamery v 3D

Martin Senderák[1]

1 Introduction

Cieľom tohto projektu počas zimného semestra je navrhnúť a implementovať čo najjednoduchšiu reprezentáciu kamier vhodnú pre priame riešenia renderovacieho problému. Toto bola moja prvá skúsenosť s týmto komplexným aspektom počítačovej grafiky, a preto bolo nevyhnutné najprv sa oboznámiť so základmi teórie renderovacej rovnice. Na prácu som využil programovací jazyk Java spolu s knižnicami určenými pre grafické zobrazenie (JavaFX).

2 Renderovacia rovnica ("Stručná" teória)

V počítačovej grafike je renderovacia rovnica integrálnou rovnicou. Táto rovnica bola nezávisle predstavená Davidom Immelom a Jamesom Kajiyou v roku 1986. Prvá Kajiho formulácia renderovacej rovnice pre počítačovú grafiku je nasledovná:

$$I(x,x') = g(x,x') \left[e(x,x') + \int_S \rho(x,x',x'') I(x',x'') \, dx'' \right], \tag{1}$$

kde:

- $I(x,x')$ predstavuje intenzitu svetla prúdiaceho z bodu x' do bodu x
- $g(x,x')$ reprezentuje vzájomnú viditeľnosť a relatívne orientácie povrchov v bodoch x a x'
- $e(x,x')$ reprezentuje svetlo emitované z x' do x.
- $\rho(x,x',x'')$ reprezentuje svetlo rozptýlené z x'' do x na povrchovom bode x'
- $\int_S dx''$ znamená integráciu cez všetky povrchové body x'' (S označuje zjednotenie všetkých povrchových bodov)

Renderovacia rovnica popisuje „množstvo svetla“ ako žiarenie emitované z bodu x v smere ω, t.j. svetlo emitované z x v smere ω plus prichádzajúce svetlo zo všetkých smerov ω' do bodu x odrazené do smeru ω materiálom. Fakt, že musíme zohľadniť všetky prichádzajúce smery, je dôvodom, prečo je potrebná integrácia. Rovnica 2 môže byť prepísaná na kompaktnejšiu uhlovú formu:

[1] senderak6@uniba.sk

$$L(x,\omega) = L_e(x,\omega) + \int_\Omega \rho(x,\omega',\omega) L_0(x,\omega') \cos(\theta') d\omega'$$

kde θ' je uhol medzi smerom ω' a normálou povrchu v bode x. Integrácia prebieha cez všetky diferenciálne pevné uhly (úzke kužele) okolo smerov ω' smerujúcich k x, v jednotkovej sfére Ω so stredom v bode x. Svetelná intenzita $L(x,\omega)$ reprezentuje svetlo opúšťajúce bod x smerom ω a $\rho(x,\omega',\omega)$ reprezentuje podiel svetla prichádzajúceho smerom ω' k x a potom odrazeného (alebo refraktovaného) do smeru ω.

2.1 Definícia renderovacieho problému

Renderovací problém je formálne definovaný ako štvorica

$$(S, \rho, L_e, W_e)$$

kde:

- S popisuje geometriu povrchu (tvary objektov vo vstupnej scéne)
- ρ je obojsmerná distribučná funkcia, ktorá popisuje materiály povrchov (odraz a priepustnosť svetla na objektoch)
- L_e popisuje svetelné zdroje (svetlo vyžarované z povrchov)
- W_e popisuje kameru (foto-citlivé senzory)

Riešením renderovacieho problému je odpoveď senzorov kamery. Spomínaná štvorica je vstupom do algoritmu, ktorý rieší renderovací problém. Renderovacie rovnice sprostredkovávajú interakcie svetla s povrchom a sú jadrom renderovacích algoritmov.

2.2 Camera

Kamera alebo meracie zariadenie je tvorené senzormi, ktoré sú zvyčajne usporiadané do mriežky a sú citlivé na svetelnú intenzitu. Každý senzor je popísaný funkciou citlivosti $w_e(x,\omega')$, ktorá vráti 1, ak svetlo v bode x smerom ω' priamo dosiahne senzor, a 0 inak. Všetky senzory v scéne sú matematicky popísané jedinou funkciou $W_e(x,\omega')$.

3 Implementácia konkrétnych kamier pre renderer využívajúci Bidirectional path tracing algoritmus

3.1 Bidirectional path tracing

Tento algoritmus generuje 'shooting' a 'gathering' trajektórie a potom ich kombinuje, aby vypočítal príspevok tejto kombinovanej trajektórie k vyrenderovanému obrazu. Keď sú trajektórie ukončené, body na scéne, kde sa 'gathering' a 'shooting' trajektórie pretínajú, sú deterministicky spojené. Toto deterministické spojenie zahŕňa konverziu 'shooting' trajektórie na 'gathering' trajektóriu, čo nahradí prichádzajúce smerovanie odchádzajúcim smerovaním.

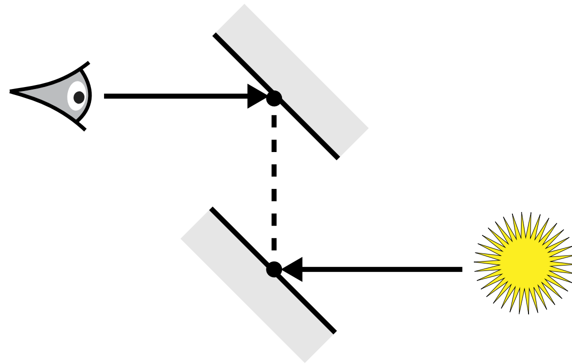

Figure 1: Bidirectional path tracing

3.2 Abstraktná kamera

Implementácia všeobecnej kamery:

```
public abstract class Camera
implements java.io.Serializable {
    public CameraBeam getNextBeam(){}
    public void watchBeam(Beam beam){}
}
```

Implementácia triedy CameraBeam je pomerne priamočiara:

```
public class CameraBeam {
    public Vector3 origin;
    public Vector3 direction;
}
```

3.3 Implementácia konkrétnych kamier

Zatiaľ som implementoval jednu jednoduchú všeobecnú kameru, ktorá využíva abstract camera class ktorá je spomenutá vyššie.

3.3.1 Jednoduchá kamera

```
public class SimpleCamera extends Camera{
    private Vec3 position , direction ;

    public SimpleCamera(Vector3 from, Vector3 to, int pixelwidth, int pixelheight,
                        double AngleX, double AngleY) {
        direction = to;
        position = from;
        w = pixelwidth;
        h = pixelheight;
        Aw = AngleX;
        Ah = AngleY;
    }
```

```
    public CameraBeam getNextBeam() {
        Vector3 poz = new Vector3(position.x, position.y, position.z);
        Vector3 dir = new Vector3(direction.x, direction.y, direction.z).normalize();

        double rot2 = (rndrAY.nextDouble() * 360);
        dir = rotateVectorCC(dir, new Vector3(direction.x, direction.y,direction.z)
            .normalize(), Math.toRadians(rot2));

        return new CameraBeam(poz, dir);
    }

    public void watch(Beam b) {
        boolean r = watch(b.origin, b.direction, b.lambda);
        if (r && lasthitspds != null) {
            spdsingle.setLambda((int) b.lambda);
            spdsingle.setY(b.power);
            lasthitspds.inc(col.SPDtoXYZ(spdsingle));

            double newY = (lasthitspds).spdshits *
            (b.source.getPower() / b.source.getNumberOfBeams());
            lasthitspds.setY(newY);
        }
    }
}
```

Poznámka: Triedy popísané tu sa "mierne" líšia od skutočnej implementácie v projekte, ale rozdiely nie sú významné.

4 Testy

Na koniec som spravil aj testy s týmto modelom kamery integrovaným do rendering engine(u), ktorý je špeciálne implementovaný pre účel tohoto projektu. Tento rendering engine nie je schopný zvládnuť ľubovoľnú geometriu, materiály a ani zatiaľ nie je možné s ním renderovať ľubovoľné scény – ale dobre slúži ako tester implementácie kamier s dvoma vstupnými scénami. V prvej scéne je biele svetlo z bodového svetelného zdroja ktorý sa "pozerá" na kameru. V druhej scéne je tiež biele svetlo z bodového svetelného zdroja a kamera, pričom kamera aj svetelný zdroj sú v rovnakom bode a "pozerajú" sa na rovnaké miesto. Pred kamerou sú tri steny, na jednej z nich je vertikálne priložená daľšia menšia stena ktorej účelom je simulovať tieň.

4.1 Simple renderer

Tento jednoduchý renderer využíva Bidirectional path tracing a teda generuje náhodné "shooting walks" a "gathering walks" funkciou next() a deterministicky ich spája. Funkcia next() je volaná, až kým nie je dosiahnutá dostatočná vizuálna kvalita, potom je obraz zbieraný z virtuálnej kamery.

Pseudo kód funkcie next():

```
function next() {
    while (iteration < MAX_ITERATIONS) {
```

```
        if (no intersection with closestTriangle) {
            if (cameraBeamActive) {
                informAllCameras(beam);
            }
            return;
        }

        updateBeamAndCameraIntersections(beam, cameraBeam);
        updateBeamDirectionTowardsCamera(beam, cameraBeam);
        iteration++;
    }

    checkAndUpdateBeamForCameras(beam, cameraBeam, cameraIntersections);
}
```

Poznámka: Funkcia popísaná tu sa "mierne" :) líši od skutočnej implementácie v projekte. Lebo ináč by ten kód bol na 2 strany.

4.2 Scéna 1.

4.3 Scéna 2.