דו"ח פרויקט

מיכל סרור 212329221
עדינה לב 566021464

רקע כללי של הפרויקט

המטרה של הפרויקט היא לייצר מצלמה שמסוגלת לתפוס סצנה מורכבת מגופים שונים ותאורה. הפרויקט נכתב בשפת Java והשתמשנו ב–IntelliJ. הפרויקט נכתב על כללי הנדסת תוכנה שלמדנו בקורס התיאורטי.

Primitives
התיקייה הזאתי מכילה המרכיבים הבסיסיים שהפרוקט נבנה עליה.

Geometries
התיקייה הזאתי מכילה הצורות המרכיבות את התמונה שלנו.

Lighting
התיקייה הזאתי מכילה התאורה שמאירה את הסצנה שלנו.

Renderer
התיקייה הזאתי מכילה הכלים העקרונים להפקת תמונה מהמצלמה.

Unittests
התיקייה הזאתי מכילה כל הבדיקות ויצירת תמונות מבחן במהלך הפרויקט.

שיפור התמונה: Glossy Surfaces/Diffuse Glass

הבעיה
הבעיה היא שבתמונה המשטחים מבריקים ובהירים מדי, והזכוכית הזו שקופה לחלוטין. זה אינו מציאותי.

השיפור
כדי לפתור את התכונה הלא מציאותית הזו, השתמשנו ב-Super-sampling. יצרנו פונקציה שתיצור אלומה של קרניים המקיפה קרן מרכזית. יצרנו גם פונקציה שתחשב את הצבע של אותה אלומת קרניים. ואז, בכל פעם שרצינו לשלוח קרן reflecton או refraction, יצרנו אלומה של קרניים מסביב לקרן זו, ושלחנו את האלומה במקום. הצבע שנבחר חושב על ידי הפונקציה החדשה שמחשבת את צבע אלומה הקרניים.

מימוש

במחלקת RayTracerBasic הוספנו שדות:

boolean superSamplingON – להפעיל\לכבות supersampling

int superSamplingGridSize – כדי להגדיר את גודל הרשת שאליו אנו יורים את הקרניים

int numOfSSRays – אנחנו יורים super-sampling כמה קרני

גם במחלקת RayTracerBasic שינינו את הפונקציה CalcGlobalEffects השניה (פונקציה שעוזר לייצר קרני reflection/refraction) כך שאם Super-Sampling מופעלת אז היא תשלח אלומה של קרני reflection/refraction במקום רק קרן יחידה.

```
/**
 * Calculates the global effects (reflection and refraction) at an intersection point.
 * @param gp The GeoPoint representing the intersection.
 * @param ray The incident ray.
 * @param level The recursion level.
 * @param k The transparency factor.
 * @return The color resulting from the global effects.
 */
1 usage   Adina Lev *
private Color calcGlobalEffects(GeoPoint gp, Ray ray, int level, Double3 k) {
    Color color = Color.BLACK;
    Vector n = gp.geometry.getNormal(gp.point);
    Material material = gp.geometry.getMaterial();
    Double3 kkr = k.product(material.kR);
    Double3 kkt = k.product(material.kT);

    if (!(kkr.lowerThan(MIN_CALC_COLOR_K))) {
        color = color.add(calcGlobalEffects(constructReflectedRay(ray.getDir(), gp.point,n), level, material.kR, kkr));
    }

    if (!(kkt.lowerThan(MIN_CALC_COLOR_K))) {
        color = color.add(calcGlobalEffects(constructRefractedRay(gp.point,ray,n), level, material.kT, kkt));
    }
    return color;
}
```

```java
/**
 * Recursive helper method for calculating global effects (reflection and refraction).
 *
 * @param ray    The incident ray.
 * @param level  The recursion level.
 * @param kx     The transparency factor.
 * @param kkx    The product of the transparency factor and the material's reflection or refraction factor.
 * @return The color resulting from the global effects.
 */
2 usages  Adina Lev
private Color calcGlobalEffects(Ray ray, int level, Double3 kx, Double3 kkx) {
    // Find the closest intersection point between the ray and scene objects
    GeoPoint gp = findClosestIntersection(ray);

    // If super sampling is enabled, shoot multiple rays (beam) and calculate the color
    if (superSamplingON == true) {
        List<Ray> beam = shootBeam(ray);
        Color beamColor = beamCalcColor(beam);

        // If there is no intersection point, return the background color;
        // otherwise, calculate the color recursively and add the beam color, then scale it by the transparency factor
        return (gp == null) ? scene.getBackground() : add(calcColor(gp, ray, level: level - 1, kkx), beamColor).scale(kx);
    }

    // If super sampling is not enabled, calculate the color recursively and scale it by the transparency factor
    return (gp == null) ? scene.getBackground() : calcColor(gp, ray, level: level - 1, kkx).scale(kx);
}
```

```java
/**
 * calculate the color
 * @param beam list of rays
 * @return the average of all the colors sent in
 */
1 usage  Adina Lev *
private Color beamCalcColor(List<Ray> beam) {
    Color result = Color.BLACK;

    //Color result = this.scene.getAmbientLight().getIntensity();

    // calculate the color of each ray and add them all together
    for (int i = 0; i < beam.size(); i++) {
        GeoPoint gp = findClosestIntersection(beam.get(i));

        //if it does not hit anything added the background
        if(gp == null) {
            result = result.add(scene.getBackground());
        }

        //else add the color of the first object it hits
        else {
            result = result.add(gp.geometry.getEmission());
        }
    }

    // divide the color by the total number of rays to get the average color
    return result.reduce(numOfSSRays);
}
```

```java
private List<Ray> shootBeam(Ray mainRay) {
    Point mrP0 = mainRay.getP0();
    List<Ray> beam = new LinkedList<Ray>();

    // get to the center of the target area (square)
    Point center = mrP0.add(mainRay.getDir());

    double halfSize = superSamplingGridSize / 2;
    // get the left corner point and use it as a basis of where to start from
    Point startingPoint = center.add(new Point(halfSize,halfSize, z: 0));
    double intervalHelper = superSamplingGridSize/(Math.sqrt(numOfSSRays));

    // shoot rays in the shape of a grid
    for (int i = 0; i < Math.sqrt(numOfSSRays); i++) {
        for (int j = 0; j < Math.sqrt(numOfSSRays); j++) {
            Point interval = new Point( x: i*intervalHelper,  y: j*intervalHelper,  z: 0);

            // add the interval to the startingPoint to get a point in the grid
            Point gridPoint = startingPoint.add(interval);

            // create the ray for this new spot in the grid
            Ray shootRay = new Ray(mrP0, gridPoint.subtract(mrP0));

            // add this ray to the list of rays to shoot
            beam.add(shootRay);
        }
    }
    return beam;
}
```