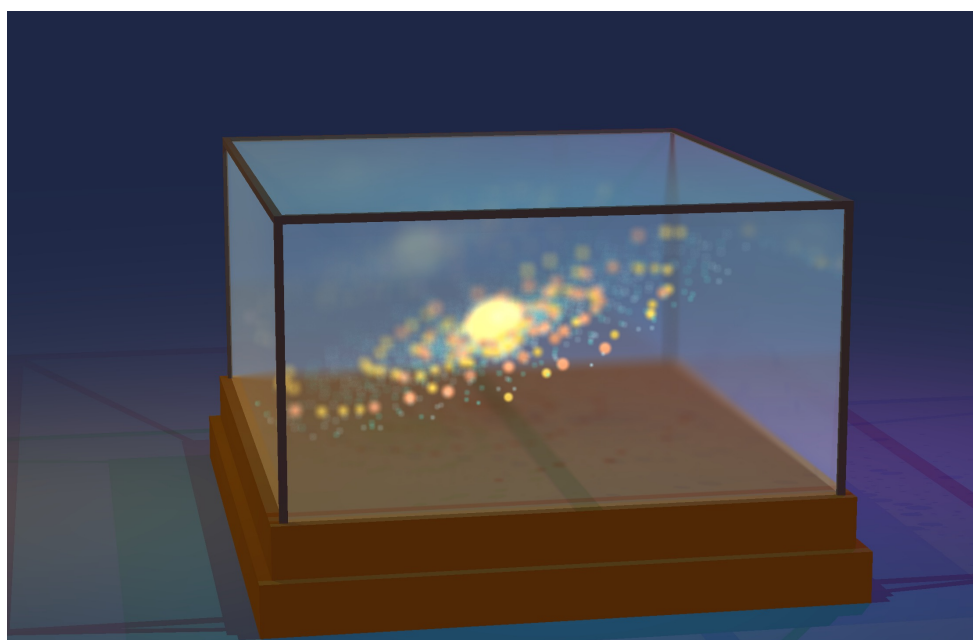


MINI PROJECT - INTRODUCTION TO SOFTWARE ENGINEERING



Introduction:

In this report we'll show the way and methodologies we used to create the final version of the program and the project image (shown above). We were tasked with creating a visually appealing picture of our choosing. After some thoughts and a few small attempts, we landed on the idea of a galaxy floating in a glass box, drawing inspiration from our love for outer space and surreal imagery.

To get the final result, we used multiple light sources around the scene with different colors that clash with the metallic frame of the box, creating visually pleasing shadow patterns; diffusive glass for the box to give the galaxy a soft blurry look; and hundreds of spheres of varying sizes and colors to make our galaxy feel real.

The way we arranged the stars is by calculating their polar coordinates and transforming them to cartesian coordinates. We had a few changeable variables in order to modify the number of arms, their density and the number of stars in each arm, and then used a function to tilt the result on its side.

To sum it up, we used the knowledge we learned in the theoretical Course and used it in practice to achieve a complex and creative image, rendered by a program that is meticulously crafted according to the standards and design patterns we learned.

Mini Project 1 - Image Improvement

The first self-learned and self-implemented task, was an image enhancement. We were given four different choices, all of them are of the category of Super-Sampling. We went with the Diffused Glass and Glossy Surfaces for our project. The objective is to simulate the way rays of light pass through realistic, diffusive glass, or reflect from glossy surfaces.

We achieve this phenomenon, by casting multiple reflected or refracted rays from the intended surfaces at every point of intersection with a previously casted ray.

In other simpler words, the camera casts one ray for every pixel in the image. When one of those rays hits a diffusive glass or a glossy surface, it turns into a beam of rays that will refract or reflect respectively. Those rays, can still hit other similar surfaces and turn into another beam of rays - thus our function here is a recursive one. In the end, the color returned by the ray, will be calculated by the beam of rays that returns the data to the function. Then, the function will calculate the average of the colors gathered by the beam of rays, and return it.

The result makes our glass in the image more realistic, and gives the impression of a more natural looking galaxy and bright stars.

The Javadoc and declaration:

```
/**
 * Generate a beam of rays from a point in a direction
 * the function works in the way that it's a square that holds a circle from edge to edge.
 * the function creates a grid of rays in the square and checks if they're inside the circle.
 * the numEdgeSamples doesn't mean how many rays inside the circle, but how many rays on the edge of the square
 * a circle is  $\pi/4$  the area of a square that holds it from edge to edge
 * that equals to about 78.5398% of the area of the square, (so about 78% of the rays will be inside the circle)
 * that means it will return  $\sim 0.785 * \text{numEdgeSamples}^2$  rays
 * @param ray the main ray
 * @param n the normal to the surface
 * @param distance the distance from the point to the center of the circle
 * @param radius the radius of the circle
 * @param numEdgeSamples the number of rays on the edge of the square the holds the circle
 * @return the list of rays that are the beam
 */
public static List<Ray> generateRayBeam(Ray ray, Vector n, double distance, double radius, int numEdgeSamples) {
```

The implementation:

```

public static List<Ray> generateRayBeam(Ray ray, Vector n, double distance, double radius, int numEdgeSamples) {
    // Check if the normal is valid
    if (n == null)
        throw new IllegalArgumentException("Normal cannot be null");
    // Check if the number of edge samples is valid
    if (numEdgeSamples < 1)
        throw new IllegalArgumentException("Number of edge samples must be at least 1");
    // Check if the distance and radius are valid
    if (distance < 0)
        throw new IllegalArgumentException("Distance cannot be negative");
    // Check if the radius is valid
    if (radius < 0)
        throw new IllegalArgumentException("Radius cannot be negative");

    List<Ray> rays = new LinkedList<>();
    rays.add(ray);
    // If the radius, distance or number of edge samples is zero, return the main ray
    if (isZero(radius) || isZero(distance) || numEdgeSamples == 1) {
        rays.add(ray);
        return rays;
    }

    Vector v, dir = ray.getDirection();
    Point head = ray.getHead();
    double gridSpacing = radius * 2 / numEdgeSamples;
    double x, y, radiusSquared = radius * radius;
    Point randomPoint, centerCircle = head.add(dir.scale(distance));
    // the 2 vectors that create the virtual grid for the beam
    Vector nX, nY;
    // if the direction of the ray is Y, the nX will be X and nY will be Z
    if (dir.equals(Vector.Y)) {
        nX = new Vector(x: 1, y: 0, z: 0);
        nY = new Vector(x: 0, y: 0, z: 1);
    } else { // if the direction of the ray is not Y, the nX will be the cross product of the direction and Y
        nX = dir.crossProduct(Vector.Y).normalize();
        nY = dir.crossProduct(nX).normalize();
    }

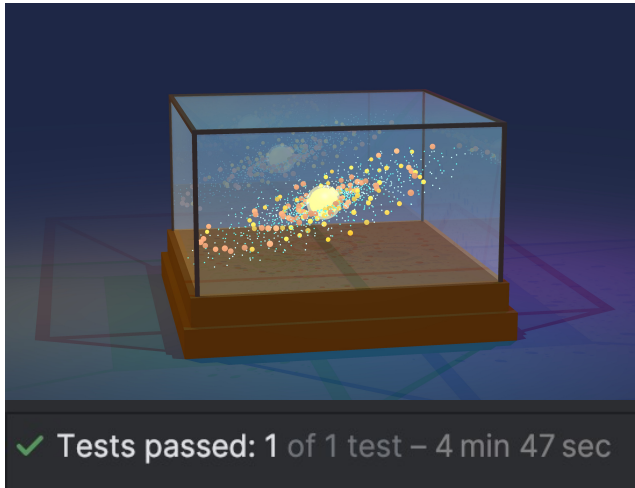
    double nd = n.dotProduct(dir); // Dot product of normal and original ray direction
    //calculating each x and y in the grid and checking if they're inside the circle/
    //if they are, we randomize them a bit inside their "zone" and then it creates the ray
    for (int i = 0; i < numEdgeSamples; i++) {
        for (int j = 0; j < numEdgeSamples; j++) {
            // Calculate the x and y coordinates in the grid
            x = (i - numEdgeSamples / 2d) * gridSpacing;
            y = (j - numEdgeSamples / 2d) * gridSpacing;

            // Check if the point (x, y) lies within the circle of given radius
            if (x * x + y * y ≤ radiusSquared) {
                x += (Math.random() - 0.5) * gridSpacing;
                y += (Math.random() - 0.5) * gridSpacing;
                try {
                    randomPoint = centerCircle.add(nX.scale(x)).add(nY.scale(y));
                    v = randomPoint.subtract(head).normalize();
                    double nv = n.dotProduct(v); // Dot product of normal and new ray direction
                    if (nv * nd > 0)
                        rays.add(new Ray(head, v));
                } catch (Exception e) {
                    j--; // Retry the same point, randomization can cause the point to become Vector.ZERO
                }
            }
        }
    }

    return rays;
}

```

Without diffusive glass:



With diffusive glass:



While achieving impressive output, this resulted in a much longer rendering time: we went from nearly 5 minutes to over an hour.


Mini Project 2 - Optimization improvement

With longer rendering times, it was time to implement the second improvement in our project: optimization. After some discussions with our lecturer about the way we should go about doing this, We went with Bounding Volume Hierarchy (BVH from now on). Keeping in mind the RDD principle, we decided to create two classes: Bounding Box in the Geometries package, and BVH in the Renderer package. We also added Multithreading bro no cap.

Multi-Threading

From the two possible ways of implementing Multithreading (MT for short), we went with the multithreading methods provided by Java's Parallel Streams. We let the different threads in the system to handle the ray casting, in order to improve the efficiency of the CPU, utilizing its power to the maximum.

```
public Camera renderImage(){ 20 usages  tomi832 +1
    final int Nx = imageWriter.getNx();
    final int Ny = imageWriter.getNy();
    if (isParallel) {
        IntStream.range(0, Ny).parallel()
            .forEach(i → IntStream.range(0, Nx).parallel()
                .forEach(j → castRay(Nx, Ny, j, i)));
    } else {
        IntStream.range(0, Ny)
            .forEach(i → IntStream.range(0, Nx)
                .forEach(j → castRay(Nx, Ny, j, i)));
    }
    return this;
}
```



Axis-Aligned Bounding Box

The Bounding Box class is the Conservative Boundary Region acceleration method, in which we create an Axis-Aligned Bounding Box for every geometry in the scene. The method creates a box around every geometry in the scene, which will help us with ray intersection points calculations like so:

Previously, whenever a ray was cast in the scene, if we wanted to check whether it intersects with a geometry in the scene, we had to run different calculations based on the geometry, varying in complexity. The fact that these calculations happened with every ray for every geometry in the scene, with most rays not even hitting more than a few objects, made the rendering time much much longer.

With CBR we can speed up the calculations by checking if a ray first intersects with it (which is much faster to calculate than any other geometry), and if it does intersect with it, we check if it intersects also with the geometry inside the box (using the regular intersection calculations as before).

```
/**
 * a method that checks if a ray intersects the box
 * @param ray the ray in question
 * @param maxDistance the maximum distance the ray can travel
 * @return true if the ray intersects the box, false otherwise
 */
public boolean intersects(Ray ray, double maxDistance) {
    Vector dir = ray.getDirection();
    Point origin = ray.getHead();
    //the arrays represent coordinates like so: [0] = x, [1] = y, [2] = z
    double[] dirValues = {dir.getX(), dir.getY(), dir.getZ()};
    double[] originValues = {origin.getX(), origin.getY(), origin.getZ()};
    double[] minValues = {minBoxPoint.getX(), minBoxPoint.getY(), minBoxPoint.getZ()};
    double[] maxValues = {maxBoxPoint.getX(), maxBoxPoint.getY(), maxBoxPoint.getZ()};

    double tmin = Double.NEGATIVE_INFINITY;
    double tmax = Double.POSITIVE_INFINITY;

    for (int i = 0; i < 3; i++) {
        if (isZero(dirValues[i])) {
            if (originValues[i] < minValues[i] || originValues[i] > maxValues[i])
                return false;
        } else {
            double t1 = alignZero(number: (minValues[i] - originValues[i]) / dirValues[i]);
            double t2 = alignZero(number: (maxValues[i] - originValues[i]) / dirValues[i]);
            if (t1 > t2) {
                double temp = t1;
                t1 = t2;
                t2 = temp;
            }
            tmin = Math.max(tmin, t1);
            tmax = Math.min(tmax, t2);
            if (tmin > tmax) return false;
        }
    }

    return tmin < maxDistance && tmax > 0;
}
```

Bounding Volume Hierarchy

We can speed up our program even more by implementing the other class we discussed, BVH. While the AABB calculations are fast, there are still a lot of them, especially when there are a lot of Geometries in the scene. That means every ray still needs to check every AABB, even if the ray is not even pointing at their direction.

To combat this, we group together different geometries that are closer to one another (e.g the triangles that make up a pyramid). Before the rendering of the scene, during the creation of the objects, the BVH would recursively go through every group of geometries, and finds the groups within them, and gives each one its own Bounding Box.

Now, when a ray checks if it intersects the objects in the scene, it recursively goes through the BVH. When there is an intersection, it goes further inside the hierarchy and checks intersections with the Bounding Boxes there (or objects in the final layer).

```
/**
 * Builds the BVH for a collection of geometries. works recursively.
 * @param geometries the collection of geometries
 */
private void buildBVH(Geometries geometries) { 2 usages  tom832
    if (geometries == null || geometries.getGeometries().isEmpty()) {
        return;
    }

    for (Intersectable geo : geometries.getGeometries()) {
        if (geo instanceof Geometries) {
            buildBVH((Geometries) geo);
        }
    }

    BoundingBox bbox = computeBoundingBox(geometries);
    geometries.setBoundingBox(bbox);
}

/**
 * @param geometries the collection of geometries
 * @return the bounding box of the geometries
 */
private BoundingBox computeBoundingBox(Geometries geometries) { 1 usage  tom832 +1
    if (geometries.getGeometries().isEmpty()) {
        return new BoundingBox(new Point(x: 0, y: 0, z: 0), new Point(x: 0, y: 0, z: 0));
    }
    //BoundingBox bbox = geometries.getGeometries().get(0).getBoundingBox();
    BoundingBox bbox = geometries.getBoundingBoxOfGeometry(index: 0);
    BoundingBox temp;
    for (int i = 1; i < geometries.getGeometries().size(); i++) {
        temp = geometries.getBoundingBoxOfGeometry(i);
        if (temp != null)
            bbox = BoundingBox.union(bbox, temp);
    }
    return bbox;
}
```

Rendering times improvements

Without any acceleration improvements:

✓ Tests passed: 1 of 1 test – 1 hr 21 min

With Multithreading (without BVH and CBR):

✓ Tests passed: 1 of 1 test – 25 min 40 sec

With MT and CBR:

✓ Tests passed: 1 of 1 test – 13 min 57 sec

With MT, CBR and BVH:

✓ Tests passed: 1 of 1 test – 6 min 47 sec

BONUSES:

- Stage 2: Normal calculations for finite cylinders
- Stage 3: Intersections with Polygon
- Stage 4: Transformation of the Camera
- Stage 6: Focused Spot light
- Stage 7: An image with 10+ geometric objects.
 - Few imaged with multiple points of view of the scene.
 - Alternative way of solving the distance to light source with shading.
- Mini-Project 1: Using Jitter for the Super-Sampling

CONCLUSION:

The project was carefully constructed, keeping as much of the design principles in mind that we learned in the theoretical course. At first it was a daunting task, but as we continued to develop the software and when we started to see the results in image form, we became more and more invested. Working in the Extreme Programming style (specifically in Pair Programming), was a refreshing way to code, boosting our enjoyment of the process and improving the work flow.

In the end, our software was fully functional and well written, resembling real life professional software.

Students:

Yosef Kornfeld 207960220

Tomer Kalman 314837121