Hillel Pinto: 1475638
Yona Szmerla: 339622375

בס"ד

Introduction to computer engineering

Mini – Project 2

Introduction:

After having done some improvement to our ray tracing model and created some images, we found out that the improvements and the model we use to create the image by using the ray tracing method is doing it very long to create an image.

In fact, we create a lot of rays, and we trace everyone of them which is bringing out we need a lot of operations and that is what is slowing the image creation.

There are 2 main problems we need to try to improve:

First, we need to try to use the maximum power the computer can allocate to our program, which can be effective for getting a better execution time for our image processing. This improvement will be done by using many threads and trying to get the maximum speed we can get to compute the image.

Secondly, the fact that we made the improvement of Anti-Aliasing to our project needs to be taken in count. Which means that according to this method, we are tracing a lot of rays in each pixel (n *m) instead of only one ray for the center of the pixel. This improvement helps us to get a more “normal” picture, but it makes the process of ray tracing very long. We will need to find a method to operate an Adaptive Super-Sampling Anti-Aliasing method, which means we will only trace more rays in a pixel if this pixel’s color seems to get a color that is not only the color of the pixel’s center.

Threads:

We need to change the way our rendering method works in order to make it be able to work by using multiple threads and not only one thread.

One of the difficulties of this method will be to synchronize this all work altogether , which means , by fact, that we need to make sure every pixel is properly traced and colored, and that this action is only made once.

We decided to use an helper class that will implement a single Pixel, this in order to have all the data we need when we send it to a different thread , and in order to be able to implement a method nextP which will give us the next pixel to be rendered. (this function will be synchronized and that will give us the surety we need).

```
private class Pixel {
    private long _maxRows = 0;
    private long _maxCols = 0;
    private long _pixels = 0;
    public volatile int row = 0;
    public volatile int col = -1;

   private synchronized void nextP(Pixel target) {
                ++col;
                ++_counter;
                if (col < _maxCols) {
                        target.row = this.row;
                        target.col = this.col;
                        }
                ++row;
                if (row < _maxRows) {
                        col = 0;
                }
}
```

After that we changed the rendering function in order to be able to render the image with threads:

```
/**
 * This function renders image's pixel color map from the scene included
with
 * the Renderer object
 * @param opt Option of rendering
 * @param isSoftShadows is rendering with soft shadows improvement
 */
private void renderImage(Options opt, boolean isSoftShadows) {
```

```
    final int nX = _imageWriter.getNx();
    final int nY = _imageWriter.getNy();

    final Pixel thePixel = new Pixel(nY, nX);

    // Generate threads
    Thread[] threads = new Thread[_threads];
    for (int i = _threads - 1; i >= 0; --i) {
        threads[i] = new Thread(()-> {
            Pixel pixel = new Pixel();
            while (thePixel.nextPixel(pixel)) {
                //construct ray for every pixel
                Ray myRay = _camera.constructRayThroughPixel(
                        _imageWriter.getNx(),
                        _imageWriter.getNy(),
                        pixel.col,
                        pixel.row);

                    _imageWriter.writePixel(pixel.col,
pixel.row,_rayTracer.traceRay(myRay, isSoftShadows));
                }
            }
        });
    }
    // Start threads
    for (Thread thread : threads) thread.run();

    // Wait for all threads to finish
    for (Thread thread : threads)
        try {
            thread.join();
        } catch (Exception e) {
        }
    if (_print) System.out.printf("\r100%%\n");
}
```

The function will generate threads as much as the value we choose at the beginning and the function nextP which is synchronized will give for each thread the next pixel that needs to be rendered until we have rendered every pixel of the image.

Adaptive Anti-Aliasing:

The adaptive anti aliasing (AAA) methods is taking base one the fact that we don't need to improve the rendering of a pixel that has only one color to render. Which means that if sending rays in any part of the pixel will compute the same color, in fact we can get the good result by tracing only the center of the pixel.

The only cases we need to trace more than one ray are when the pixel is computing different colors, which means that there is more than one object that need to get color in that pixel (there is change of color in the image in that pixel).

So we decided to make the work of rendering a pixel in AAA method to be recursive.

Which means that after getting the ray that is oriented from the camera to the pixel center, we will construct a HashTable<Integer,ColoredRay> which will get 5 colored rays :

- 1 : corner up left
- 2 : corner up right
- 3 : center
- 4 : corner down left
- 5 : corner down right

The ColoredRay class is a helping class that will help us to keep every ray and the color that was traced to it. The point of keeping those 2 values is because we are going to reuse those rays in the recursive calls and we do not want to trace the same ray multiple times, so we trace it ome time and we keep the value for next use.

```
public class ColoredRay {
    private Ray ray;
    private Color color;

    public ColoredRay(Ray ray, Color color) {
        this.ray = ray;
        this.color = color;
    }

    public Ray getRay() {
        return ray;
    }
```

```
    public Color getColor() {
        return color;
    }
}
```

This is the same fact that brought us to have use of HashTable which will help us to keep the rays in a structured way and to be able to get each one in O(1)

After getting those five Colored ray, we will check the distance of each one's value with the center. If one of them gives a value which distance's is more than expected, we will cut the pixel in 4 respective pixel of same size (each corner) and we will send every part as a different pixel in recursive method. This will be done until the color is the same for each 5 rays or when we get to the maximum depth of the recursion.

To be able to do all this, we needed to add 3 different functions in the camera class :

- The first one is to construct the five rays that we spoked about when we have only the ray at the center of the pixel. To be done, get the point on the view plane that is traced by the ray (the center of the pixel) by using the function in Ray class with the distance from the camera point 0 to the plane at that specific pixel (slightly different than the distance field of the camera)

```
/**
 * Function that gets a pixel's center ray and constructs a list of 5 rays
from that ray,
 * one at each corner of the pixel and one in the center
 * @param myRay the center's ray
 * @param nX number of pixel in width
 * @param nY number of pixels in height
 * @return list of rays
 */
public List<Ray> construct5RaysFromRay(Ray myRay, double nX, double nY) {

    List<Ray> myRays = new LinkedList<>();

    //Ry = h / nY - pixel height ratio
    double rY = alignZero(_height / nY);
    //Rx = h / nX - pixel width ratio
    double rX = alignZero(_width / nX);


    //distance from the camera p0 to the plane
    //we need to calculate the distance from the point on the camera to the
plane
```

```
    //for that we use the cos of the angle of the direction ray with vTo
vector
    double t0 = _distance;
    double t = t0 / (_vTo.dotProduct(myRay.get_dir())); //cosinus on the
angle
    Point3D center = myRay.getPoint(t);

    //[-1/2, -1/2]
    myRays.add( new Ray(_p0, center.add(_vRight.scale(-rX /
2)).add(_vUp.scale(rY / 2)).subtract(_p0)));
    //[1/2, -1/2]
    myRays.add( new Ray(_p0, center.add(_vRight.scale(rX /
2)).add(_vUp.scale(rY / 2)).subtract(_p0)));

    myRays.add(myRay);
    //[-1/2, 1/2]
    myRays.add( new Ray(_p0, center.add(_vRight.scale(-rX /
2)).add(_vUp.scale(-rY / 2)).subtract(_p0)));
    //[1/2, 1/2]
    myRays.add( new Ray(_p0, center.add(_vRight.scale(rX /
2)).add(_vUp.scale(-rY / 2)).subtract(_p0)));
    return myRays;
}
```

- The second one helps us to get the ray to the center of a pixel when we have the down right corner's ray of that pixel. It will be in use to get the ray through the centers of the sub-pixels we will send to render in the recursive method:

```
-   /**
     * Construct ray through nthe center of a pixel when we have only the
    bottom right corner's ray
     * @param ray the ray
     * @param nX number of pixel in width
     * @param nY number of pixels in height
     * @return center's ray
     */
    public Ray constructPixelCenterRay(Ray ray, double nX, double nY) {

        //Ry = h / nY - pixel height ratio
        double height = alignZero(_height / nY);
        //Rx = h / nX - pixel width ratio
        double width = alignZero(_width / nX);


        Point3D point = getPointOnViewPlane(ray);
        point = point.add(_vRight.scale(width / 2)).add(_vUp.scale(-
    height / 2));
        return new Ray(_p0, point.subtract(_p0));
    }
```

- The third will be a function that will give us 4 rays at the centers of the sides of the pixel. Those rays will be the corner's rays of the future sub pixels :

```java
/**
 * Function that returns a list of 4 rays in a pixel
 * one on the top upper the center point
 * one on the left of the center
 * one on the right of the center
 * one on the bottom of the center
 * @param ray the center's ray
 * @param nX number of pixel in width
 * @param nY number of pixels in height
 * @return list of rays
 */
public List<Ray> construct4RaysThroughPixel(Ray ray, double nX, double nY)
{

    //Ry = h / nY - pixel height ratio
    double height = alignZero(_height / nY);
    //Rx = h / nX - pixel width ratio
    double width = alignZero(_width / nX);

    List<Ray> myRays = new ArrayList<>();
    Point3D center = getPointOnViewPlane(ray);


    Point3D point1 = center.add(_vUp.scale(height / 2));
    Point3D point2 = center.add(_vRight.scale(-width / 2));
    Point3D point3 = center.add(_vRight.scale(width / 2));
    Point3D point4 = center.add(_vUp.scale(-height / 2));
    myRays.add(new Ray(_p0, point1.subtract(_p0)));
    myRays.add(new Ray(_p0, point2.subtract(_p0)));
    myRays.add(new Ray(_p0, point3.subtract(_p0)));
    myRays.add(new Ray(_p0, point4.subtract(_p0)));
    return myRays;
}
```

In order to get a point on the view plane we need to get it with the function ray.getPoint(double distance) but first we need to get that distance which changes for every point on the view plane (the distance we have in the camera field is the distance to the center pixel. In order to get the point we created a helper method in the camera that will give us the point using the cosinus of the angle between vTo and the ray's vector :

```java
/**
 * Function to find a specific point on the plane
 * we need to calculate the distance from the point on the camera to the
plane
 * for that we use the cos of the angle of the direction ray with vTo
vector
 * @param ray ray to the specific point
 * @return the distance to the point
 */
private Point3D getPointOnViewPlane(Ray ray) {
    double t0 = _distance;
    double t = t0 / (_vTo.dotProduct(ray.get_dir())); //cosinus of the
angle
```

```
    return ray.getPoint(t);
}
```

Color:

In order to check if the colors are equals or not, we implemented the equals method in our Color Wrapper

We decided that the maximum distance between 2 colors coordinates will be 10 in order for our improvement to be useful (too much won't do antialiasing at all, too few will be the same than anti-aliasing without adaptive)

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Color color = (Color) o;
    return Math.abs(this.r - color.r) < 10 && Math.abs(this.g - color.g) <
10 && Math.abs(this.b - color.b) < 10;
}
```

Enum:

We added a Enum in order to help to choose the improvements,

2 improvements can be chosen at a time:

```
/**
 * All options for render the image
 */
public enum Options {
    DEFAULT, ANTI_ALIASING, DEPTH_OF_FIELD, SOFT_SHADOWS, THREADS,
ADAPTIVE_ANTI_ALIASING
}
```

```
public void renderImage(Options opt1, Options opt2)
```

Improvement times :

- Threads:
    i. Without the threads, the teapot test has taken: 2min
    ii. With the threads it took: 1 min 20 sec

- Adaptive anti-Aliasing:
    i. Without adaptive anti-aliasing our image took: 25 min 30 sec
    ii. With adaptive anti-aliasing it took: 9 min 23 sec