Menachem Heller

Bezalel Jacober

Introduction to Software Engineering
Mini Project – Report
Part 2/2
JCT - Fall 2022

Introduction:

In this part of the report our goal is to show the improvements we made to our project focusing on the efficiency of the ray tracing process.

We did so by adding two components, first, we added multi-threading to the ray tracing process, secondly, we added an improved Anti-Aliasing method Called Adaptive Anti-Aliasing. We will elaborate on both shortly.

Threads:

up to this point we rendered our image using a single thread, to improve cpu utilization for a faster rendering time , we added multi-threading to our render image function, our main concern though is that the process of multi-threading has to be done safely so we do not have two threads tracing and coloring the same pixel, to do so we added a class Pixel.java {@credit Author Dan Zilbershtein} that is responsible on designating the next pixel to be rendered, while securing the separate threads from causing errors.

The main function in the class is the nextPixel method:

```
public boolean nextPixel() {
    synchronized (mutexNext) {
        if (cRow == maxRows)
            return false;
        ++cCol;
        if (cCol < maxCols) {
            row = cRow;
            col = cCol;
            return true;
        }
        cCol = 0;
        ++cRow;
        if (cRow < maxRows) {
            row = cRow;
            col = cCol;
            return true;
        }
        return false;
    }
}
```

which moves us to next pixel to render, but is done synchronized , assuring a correct ray tracing process.

We then adjusted our render image function to support multi-threading:

```
public void renderImage() {
    // check that image, writing and rendering objects are
instantiated
    if (imageWriter == null)
        throw new MissingResourceException("image writer is
not initialized", ImageWriter.class.getName(), "");

    if (rayTracer == null)
        throw new MissingResourceException("ray tracer is not
initialized", RayTracer.class.getName(), "");

    int nX = imageWriter.getNx();
    int nY = imageWriter.getNy();

    //initialize thread progress reporter
    Pixel.initialize(nY, nX, printInterval);
```

```
    // for each pixel (i,j) , construct  ray/rays from camera
through pixel,
    // functions use rayTracer object to get correct color,
then use imageWriter to write pixel to the file

    if (isUseDOF()) {
       renderImageDOF(nX,nY);
    }
    //cast ray according to Anti-Aliasing method set to Camera
    else {
        switch (getAntiAliasing()) {
            // no method used - cast single ray to center of
pixel
            case NONE -> {
                renderImageAntiAliasingNone(nX,nY);
            }
            // bean of random rays cast for each pixel besides
the ray towards the center
            case RANDOM -> {
                renderImageAntiAliasingRandom(nX,nY);
            }
            // four rays cast to four corners of pixel besides
the ray towards the center
            case CORNERS -> {
                renderImageAntiAliasingCorners(nX,nY);
            }
            case ADAPTIVE -> {
                renderImageAntiAliasingAdaptive(nX,nY);
            }
        }
    }
}
```

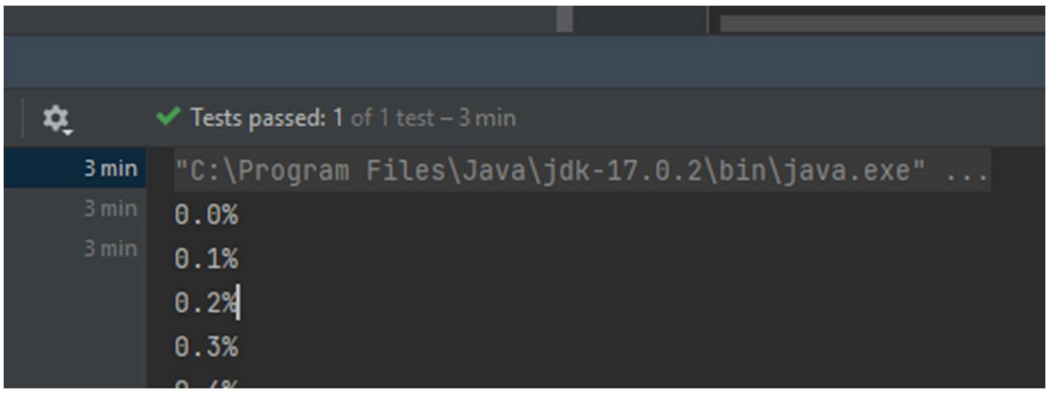
each of the inner renderImage…() functions renders the image using the ray tracing methods intended for the each type of improvement.

Runtime improvement using threads:

We rendered this image using 600x600 pixels and the adaptive anti-aliasing method with recursion level of 4 (pixel divided max to 2x2)

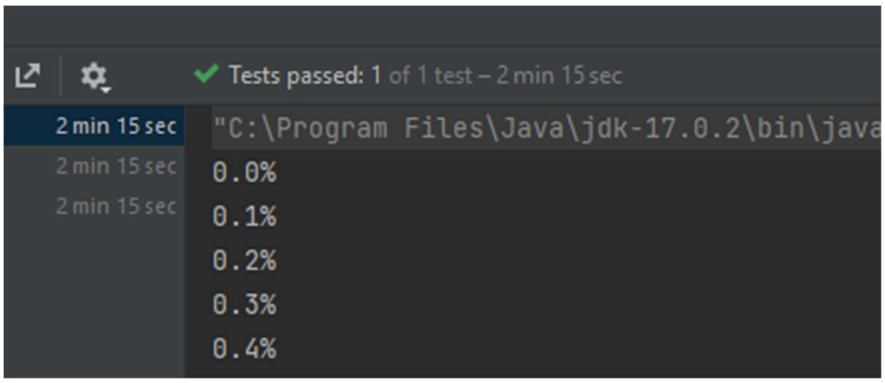
With no multi-threading – runtime:

```
ImageWriter imageWriter = new ImageWriter( imageName: "ProjectTest14",  nX: 600, nY: 600);
Camera camera = new Camera.CameraBuilder(new Point( x: 150,  y: 130,  z: 1400), new Vector( x:
        .setVPSize( width: 600,  height: 600)
        .setVPDistance(1000)
        .setImageWriter(imageWriter) //
        .setRayTracer(new RayTracerBasic(scene))
        .setAntiAliasing(AntiAliasing.ADAPTIVE).setRecurseDepth(4)
        .setMultithreading(0)
        .setDebugPrint(0.1)
        .build();//
camera.renderImage(); //
camera.writeToImage();
```



With multi-threading – using four threads:

```
ImageWriter imageWriter = new ImageWriter( imageName: "ProjectTest14",  nX: 600, nY: 600);
Camera camera = new Camera.CameraBuilder(new Point( x: 150,  y: 130,  z: 1400), new Vector( x:
        .setVPSize( width: 600,  height: 600)
        .setVPDistance(1000)
        .setImageWriter(imageWriter) //
        .setRayTracer(new RayTracerBasic(scene))
        .setAntiAliasing(AntiAliasing.ADAPTIVE).setRecurseDepth(4)
        .setMultithreading(4)
        .setDebugPrint(0.1)
        .build();//
camera.renderImage(); //
camera.writeToImage();
```



A difference of 45 seconds in runtime between using or not using multithreading.

Choosing The Improvement:

The decision of what improvement to use is extremely easy.
we will show how changing a single line in the test case will render the same image with different methods, without changing anything in the image.

This code will render the image using the adaptive method:

```
ImageWriter imageWriter = new ImageWriter("ProjectTest14",
600,600);
Camera camera = new Camera.CameraBuilder(new Point(150, 130,
1400), new Vector(-0.2, -0.15, -1), new Vector(0, (double) 20
/ 3, -1)) //
        .setVPSize(600, 600)
        .setVPDistance(1000)
        .setImageWriter(imageWriter) //
        .setRayTracer(new RayTracerBasic(scene))

.setAntiAliasing(AntiAliasing.ADAPTIVE).setRecurseDepth(16)

        .setMultithreading(3)
        .setDebugPrint(0.1)
        .build();//
camera.renderImage(); //
camera.writeToImage();
```

By switching one line:

```
 .setAntiAliasing(AntiAliasing.NONE)
```

Now the image will be rendered without any improvements.

If we want to use Depth of field improvement instead

We will remove the Anti-Aliasing setter and instead add the line:

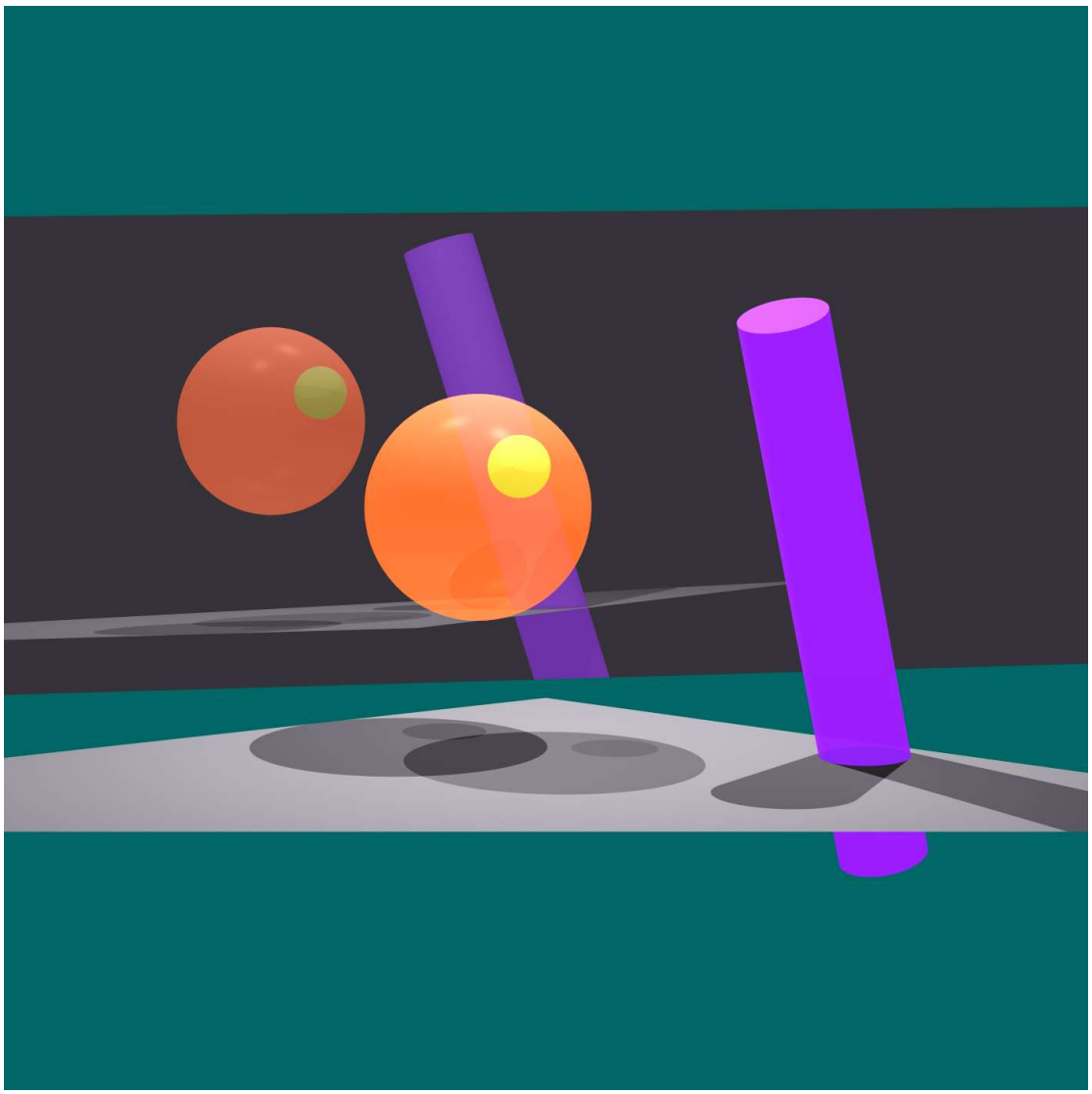
```
.setUseDOF(true).setDof(1000).setApertureRadius(1.5)
```

and so forth, allowing a very swift change between the options.a

Adaptive Anti-Aliasing:

The second improvement we added is the adaptive anti-aliasing method. the problem we encountered was, when we wanted to render images in higher resolution using the Random method we reached very long processing runtimes, for example the following picture (one of our earlier test)

That was rendered using the random method with 33x33 rays per pixel, the image has 1400x1400 pixels.



it took took 5 hours and 13 minutes to render the image!

To solve this issue, we reduce the processing time using the Adaptive method.

Explenation:
with the random method we cast n*m rays towards each pixel to sample its color regardless of the pixel or what the color is, lets look for example at the bottom of the picture above, the entire bottom section of the picture is a solid color that does not changed , the color of the pixel could be determined by the first ray, instead we're casting n*m-1 extra rays for that pixel for no reason, here is where the adaptive method interjects, instead of casting n*m rays by default , we sample the center and the four corners, if all the colors match, we set the color to the sum of the ray colors/5' there is no need for further calculation, the entire pixel has the same color.
if one of the corners does not match we recursively send the subpixel in that corner to calculate its color , this process continues until all corners match the pixel, or the max recursive depth that we set is exceeded,
by casting extra rays only for specific pixels and not the entire image by default we achieve better processing times.

The Adaptive method is implemented in our code as follows:

This is the wrapping function, it casts rays to the center and the corners and calls the recursive function to calculate a subpixel if necessary,
if all match, only the five original rays were casted and we return the color

```
private void castRayAdaptive(int Nx, int Ny, int j, int i, int
size, int depth) {

    // construct ray through pixel
    Ray ray = constructRay(Nx, Ny, j, i);
    Point center = ray.getPoint(distance);
    //construct four rays to the corners of the pixel
    //function returns list with rays sorted from top left
corner to bottom left, clockwise.
    var rayBeam = constructRayCorners(Nx, Ny, ray, size);

    // calculate  color of pixel , if the corner color matches
the center , add the color,
    // else recursively  calculate the subpixel which corner
color does not match
    // ray towards center color
    Color color = rayTracer.traceRay(ray);
```

```
    int k = 0; //index of ray in the list
    for (var r : rayBeam) {
        Color cornerColor = rayTracer.traceRay(r);
        if (color.equals(cornerColor))
            color = color.add(cornerColor);
        // corner color does not match
        else {
            // get the center point of the subpixel
            Point subPixelCenter = getSubPixelCenter(k, Nx,
Ny, size * 2, center);
            //call recursive function to calculate color of
subpixel
            color = color.add(constructRayAdaptiveRec(Nx, Ny,
subPixelCenter, r, size * 2, depth));
        }
        k++;
    }
    //get avg color of pixel between all the samples
    color = color.reduce(rayBeam.size() + 1);

    //write pixel
    imageWriter.writePixel(j, i, color);
}
```

the following recursive function uses the same methodology, but it allows the recursive call to itself, note that we end the recursion on recursion depth exceeded or if the colors of all corners of a subpixel match the color of the center.

```
private Color constructRayAdaptiveRec(int Nx, int Ny, Point
center, Ray rayToCorner, int size, int depth) {

    //if division size is smaller or equal to recursion depth
limit , continue to calculate subpixel color
    if (size <= depth) {
        //ray from camera to center of subpixel
        Vector camToSubPixel = center.subtract(p0);
        Ray ray = new Ray(p0, camToSubPixel);
        Color color = rayTracer.traceRay(ray);
        //construct four rays to the four corners of the
subpixel
        var cornersBeam = constructRayCorners(Nx, Ny, ray,
size);

        //if color of the corners matches , ad rhe color,
```

```
otherwise ,recursively calculate
        //the color of the mismatching corner subpixel

        int k = 0;
        for (var r : cornersBeam) {
            Color cornerColor = rayTracer.traceRay(r);
            if (color.equals(cornerColor))
                color = color.add(cornerColor);
            else {
                // get the center point of the sub pixel
                Point subPixelCenter = getSubPixelCenter(k,
Nx, Ny, size * 2, center);
                // recursively calculate color , doubling
division of pixel by two for every level of recursive call
                color = color.add(constructRayAdaptiveRec(Nx,
Ny, subPixelCenter, r, size * 2, depth));
            }
            k++;
        }
        //return the color of the subpixel
        color = color.reduce(cornersBeam.size() + 1);
        return color;

    // recursion depth is reached , return the color of the
corner of the subpixel
    } else {
        return rayTracer.traceRay(rayToCorner);
    }
}
```

the getSubPixelCenter() function used by both wrapper and recursive function, is used as follows: our functions know to take a center point and calculate the color of the subpixel – the size of the subpixel is determined by the recursion depth, the deeper we go , smaller the subpixel becomes. but we need to move to the correct subpixel's center position , here is where our function helps, using the index of the ray in the list of rays cast to the subpixel corners inn the current recursive call we determine which subpixel is needed for the recall , and we move the center point accordingly .
the implementation:

```
private Point getSubPixelCenter(int k, int Nx, int Ny,
int size, Point center) {

    //calculate scaling factors with division size
parameter
```

```
    double Rx = alignZero(((double) width / Nx) / size);
    double Ry = alignZero(((double) height / Ny) / size);
    Point p = center;

    switch (k) {
        case 0: // top left corner subpixel
            return
p.add(vUp.scale(Ry)).add(vRight.scale(Rx));
        case 1: //top right corner subpixel
            return p.add(vUp.scale(-
Ry)).add(vRight.scale(Rx));
        case 2: // bottom right corner subpixel
            return p.add(vUp.scale(-
Ry)).add(vRight.scale(-Rx));
        case 3: // bottom left corner subpixel
            return
p.add(vUp.scale(Ry)).add(vRight.scale(-Rx));
    }
    return null;
}
```

to save processing time we defined the Colors.java class equals() function to return true if two colors are less than 10 apart, that allows us to save runtime while the image quality will not be affected.

```
public boolean equals(Object o) {
   if (this == o) return true;
   if (!(o instanceof Color)) return false;
   Color color = (Color) o;
   return Math.abs(this.rgb.d1-color.rgb.d1)<10
         &&Math.abs(this.rgb.d2-color.rgb.d2)<10
         &&Math.abs(this.rgb.d3-color.rgb.d3)<10;
}
```

Runtime Differences Examples:

Here is a photo that has 8 geometries in it, we used 600x600 pixels and the Random anti-aliasing method with 33x33 rays cast per pixel

It took 53 minutes and 37 seconds to render.

Following, is our project picture, same number of pixels, 600x600, but we used the Adaptive anti-aliasing method instead, with a recursion level of 4 (16) (the pixel will be divided up to 8x8)
for comparison - our image has 134 geometries ,more complex shading, reflective and transparent surfaces.

Runtime:

```
ImageWriter imageWriter = new ImageWriter( imageName: "ProjectTest14",  nX: 600, nY: 600);
Camera camera = new Camera.CameraBuilder(new Point( x: 150,  y: 130,  z: 1400), new Vector( x: -0.
        .setVPSize( width: 600,  height: 600)
        .setVPDistance(1000)
        .setImageWriter(imageWriter) //
        .setRayTracer(new RayTracerBasic(scene))
        .setAntiAliasing(AntiAliasing.ADAPTIVE).setRecurseDepth(16)
        .setMultithreading(3)
        .setDebugPrint(0.1)
        .build();//
camera.renderImage(); //
camera.writeToImage();
```

```
Tests passed: 1 of 1 test – 49 min 35 sec
"C:\Program Files\Java\jdk-17.0.2\bin\java.exe" ...
0.0%
```

As we can tell, the difference in quality between the images is not significant, and for a much more complex image, we achieve shorter processing times as we wished.

Another comparison:

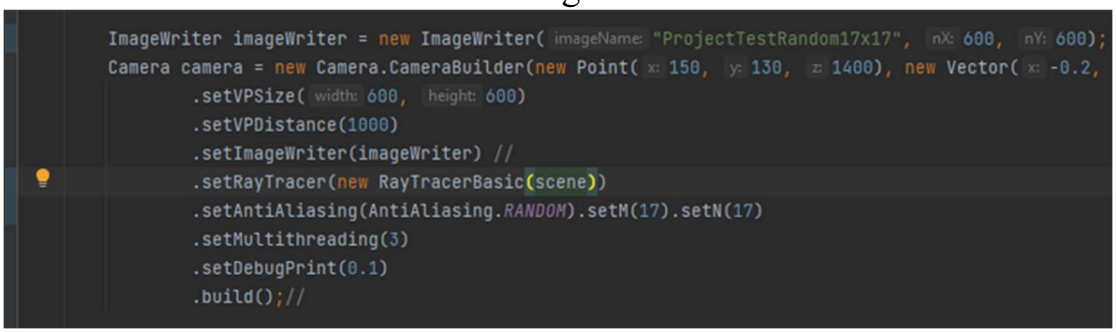
Project picture - 600x600 pixels.
Random anti-aliasing method – 17x17 rays cast per pixel.

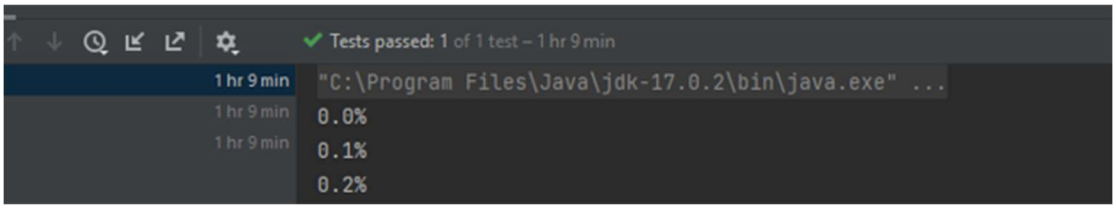
Project picture
600x600 pixels.
Adaptive anti-aliasing method recursion depth 8 (pixel divided up to 4x4)

As we can see both images are similar results in quality.

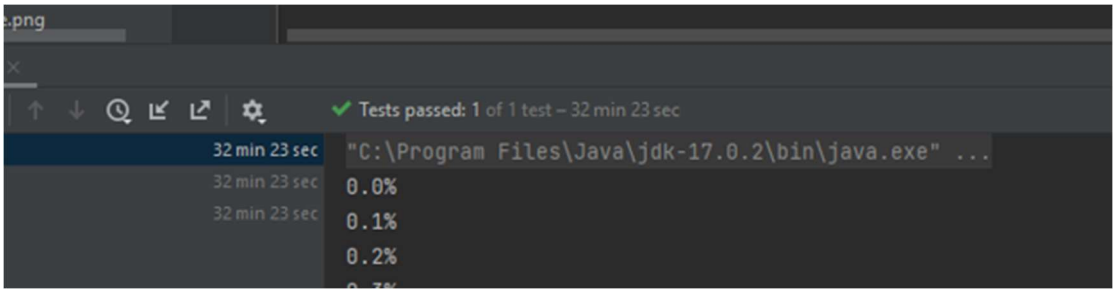
But let's look at the difference in runtime:

For the random method rendered image:





For the adaptive method rendered image:



One hour and ten minutes, compared to only thirty-two minutes. Here too, we see the vast improvement using the Adaptive method over the Random method.