Menachem Heller

Bezalel Jacober

Introduction to Software Engineering
Mini Project – Report
Part 1/2
JCT - Fall 2022

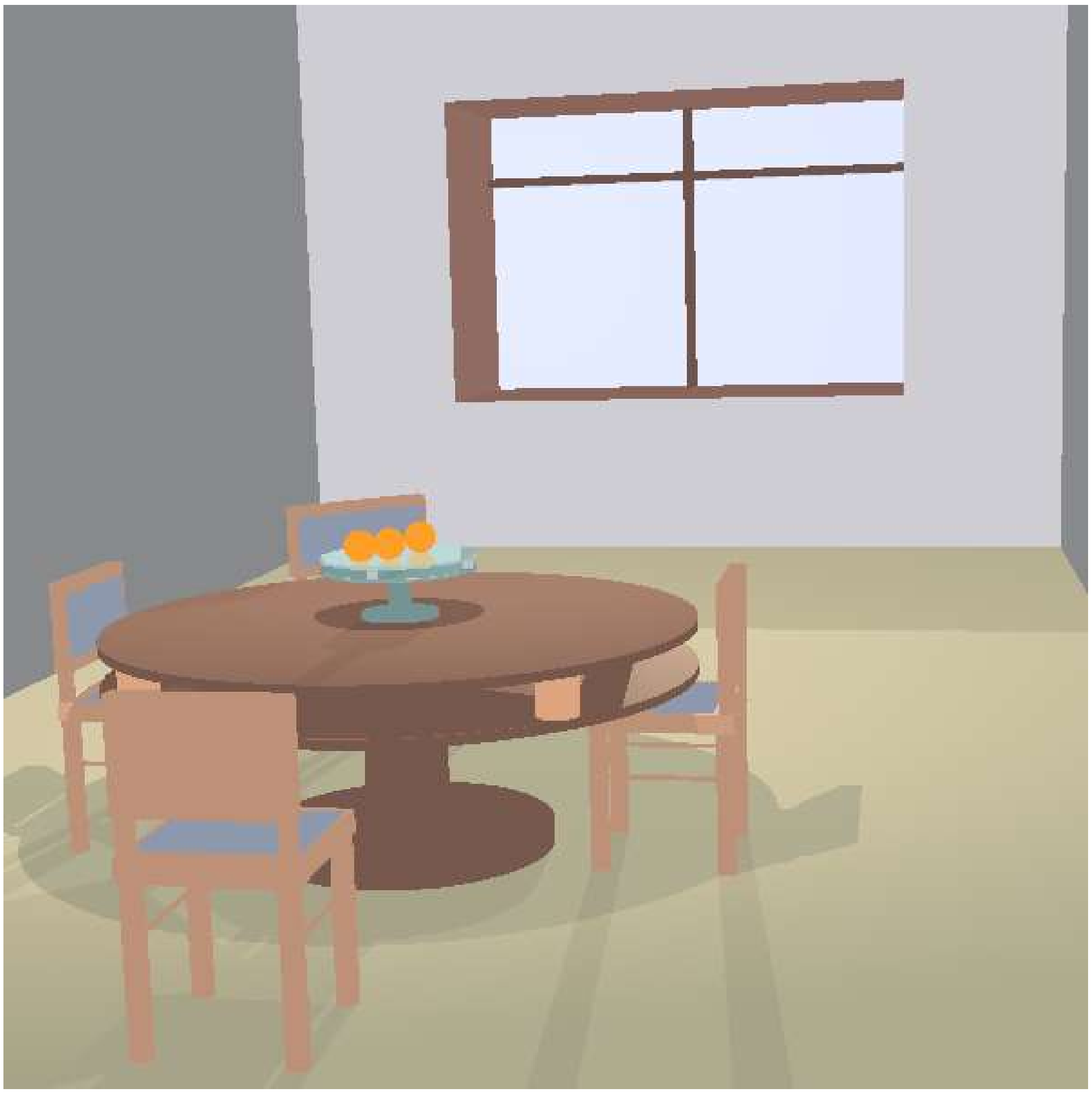
Introduction:

In this part of the report our goal is to demonstrate the methodologies we used to create our picture, and to show the two improvements we added to our project/picture.

The requirement was to create a scene that includes multiple geometries, with a few light sources added to the scene, which emphasizes the shading affects across the scene.

Our thought process in creating the picture was to integrate all the functionalities we added to the ray tracing, using reflective surfaces (Platter), transparent surfaces (Window), and cross shadows caused by the multiple light sources.

Our picture:



In our scene, we created a room, in the middle of the room we have the table, on the table we have a platter with three oranges, around the table we have a few chairs, and further behind the table and chairs, on the back wall, we added the window.

Shortly, we elaborate about the classes we added to simplify the scene construction, but first, here are general details about the scene.

The scene is constructed with 134 geometries and two light sources. one spotlight is located outside the window directed inwards towards the room, and the second spotlight is located in the ceiling in the back left corner of the room.

Chairs:

Let's take a closer look at the chairs, each chair is constructed using 22 geometries, making it very difficult to replicate, if done manually, we solved this problem creating a Chair object that we construct as follows:

```
public Chair(Point p, double seatLength,
 double height, double seatWidth,
double backWidth, double legRadius,
 double barRadius, Vector forward, Vector right, Color color, boolean
isUpsideDown)
```

using this class, we avoid the need to define twenty-two geometries with precise location and direction for each one of them, rather we use a point which is the center of the seat, a few numeric values to determine the measurements of the chair and its elements, and finally we use two vectors determining the forward direction of the chair and an orthogonal to the right vector . This way we can easily reconstruct chairs across the scene. All we need to decide is the location and direction of the chair, specify the measurements, and the class will return a full chair object.

(we added a feature with the Boolean parameter were we can decide before constructing if we want the chair to be flipped upside-down).

We added a function

```
public Geometries getGeometries() {
    return new Geometries(seatUp, seatDown,
seatSideFr, seatSideLeft, seatSideBck,
seatSideRight,backLeft, backRight,
frontLeft,frontRight, leftBar, rightBar, backrestLft,
backrestBck, backrestRt, backrestFr,
backrestTop,backrestLftPeg,backrestRtPeg,seatCover,ba
ckrestCover);
}
```

that returns the chair's elements as a composite of geometries, allowing us to add it to the scene's geometries.

Table:

For the table object also, we used the same type of thinking, although the table is created using only eight geometries, which makes it a little easier to construct manually than the chair, regardless, here also we felt a class will allow a much easier way to create tables for our scene.

The constructor:

```
 public Table(int height, double radius, Color color,
Point position, Vector dirHeight, Vector dirSurface)
```

we ask for the position point of the table (center point of the base), forward direction vector , and its orthogonal upwards direction vector, measurements for the table height and size , again, making it very easy to create or move tables in our scene.

Here to we added a function

```
public Geometries getGeometries() {
    return new
Geometries(surfaceTop,leg,base,surfaceBase,cy1,cy2,cy
3,cy4);
}
```

Setters:

In both the chair and table classes we added setters for each different part of the object allowing us to change the color and material for each specific part of the chair and table individually.

 Example of the setters for the backrest of the Chair:

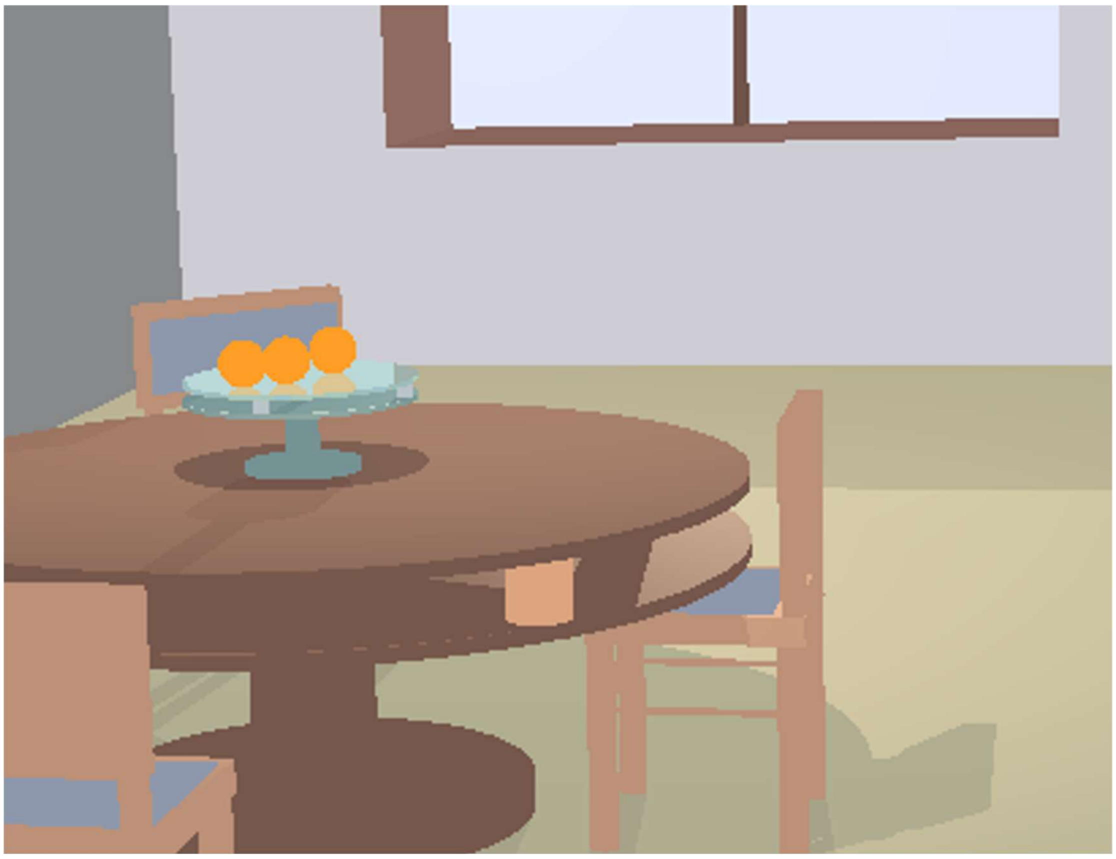
```
public Chair setBackRestEmission(Color color) {
    backrestFr = (Polygon) backrestFr.setEmission(color);
    backrestBck = (Polygon)
backrestBck.setEmission(color);
    backrestLft = (Polygon)
backrestLft.setEmission(color);
    backrestRt = (Polygon) backrestRt.setEmission(color);
    backrestTop = (Polygon)
backrestTop.setEmission(color);
    return this;

}

public Chair setBackRestMaterial(Material mt) {
    backrestFr = (Polygon) backrestFr.setMaterial(mt);
    backrestBck = (Polygon) backrestBck.setMaterial(mt);
    backrestLft = (Polygon) backrestLft.setMaterial(mt);
    backrestRt = (Polygon) backrestRt.setMaterial(mt);
    backrestTop = (Polygon) backrestTop.setMaterial(mt);
    return this;
}

public Chair setBackRestKs(double ks) {
    Material mt = backrestFr.getMaterial();
    mt = mt.setkS(ks);
    return setBackRestMaterial(mt);
}

public Chair setBackRestKd(double kd) {
    Material mt = backrestFr.getMaterial();
    mt = mt.setkD(kd);
    return setBackRestMaterial(mt);
}

public Chair setBackRestKr(double kr) {
    Material mt = backrestFr.getMaterial();
    mt = mt.setkR(kr);
    return setBackRestMaterial(mt);
}

public Chair setBackRestKt(double kt) {
    Material mt = backrestFr.getMaterial();
    mt = mt.setkT(kt);
    return setBackRestMaterial(mt);
}

public Chair setBackRestShinines(int nshinines) {
    Material mt = backrestFr.getMaterial();
    mt = mt.setnShininess(nshinines);
    return setBackRestMaterial(mt);
}
```

Improvements:

Anti-Aliasing:

Now, after the scene construction is complete lets take a closer look at the improvements we added to our project.

If we take a closer look at the scene,



Clearly, we can see on the window frame and the table edges that the objects come out very pointy giving a very unrealistic look to the scene, the table, which is a circular object, its edges should be a smooth curved line and not a rough line as we see.

This happens because we sample the center of the pixels to get its color, and we proceed to color the entire pixel with the same color, which isn’t the case all the time, the edge of the table might be covering only part of the pixel, but the entire pixel is wrongly colored in the tables color.

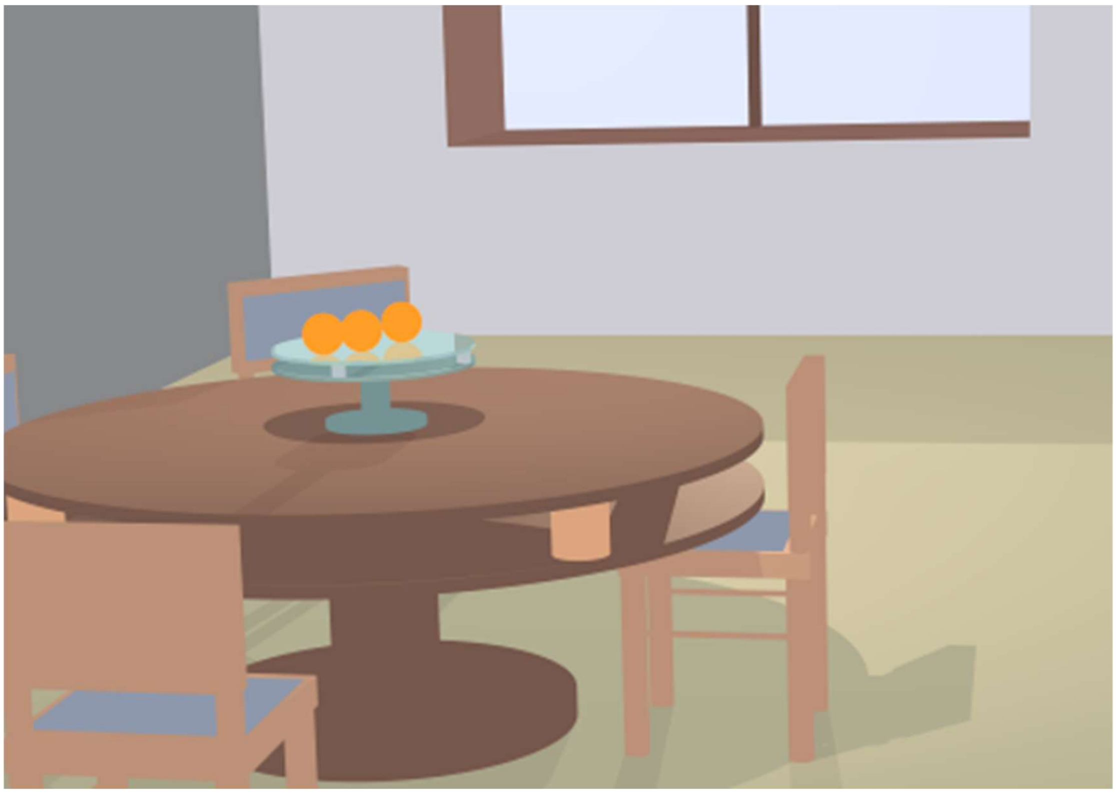
To solve this issue, we added Anti-Aliasing functionality to our camera object, which can take one of the four methods from the Enum we added to the project, to use in rendering the picture:

```
public enum AntiAliasing
   {
      NONE,

      RANDOM,

      CORNERS,

      ADAPTIVE
   }
```

In the camera we added an Antialiasing field with a setter , allowing us to easily change between the methods.

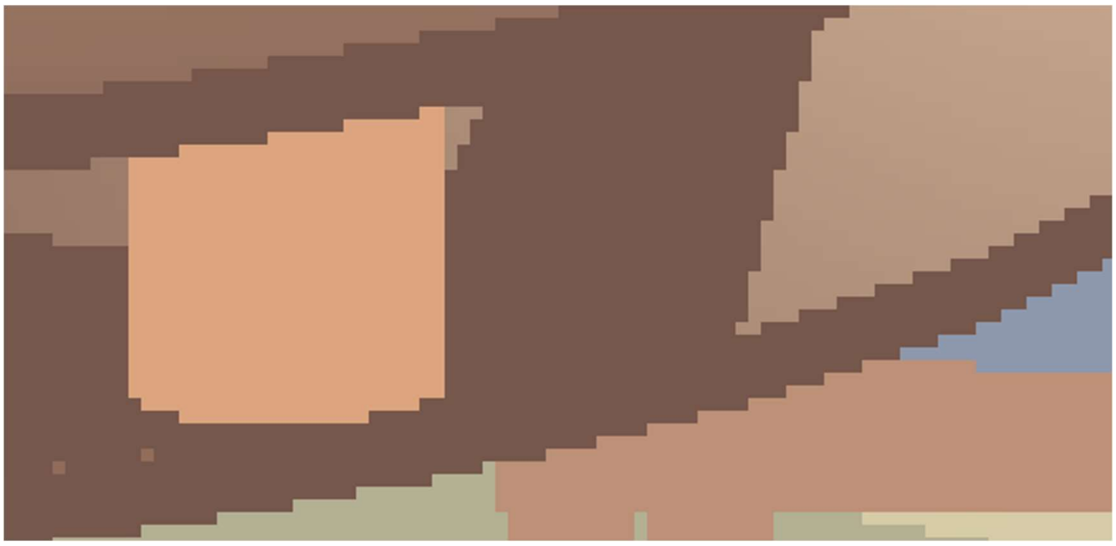
We can either use no anti-aliasing method, the Corners method, which samples the center of the pixel plus the four corners and returns the mean value of all the colors combined, or we can use the method that produces the best results, the Random method, where we create an n*m sub grid within each pixel and cast n*m rays for each pixel through each of the sub grids cells, and calculate the mean value of the color using all the samples. (N and M are also fields in the camera object with setters allowing easy interchange between values).

Using these methods gives a more mixed color for a pixel that contains a few colors giving a much "smoother" look to the scene as we can see in the identical snapshot taken from the scene rendered using the random method with 17x17 rays per pixel:

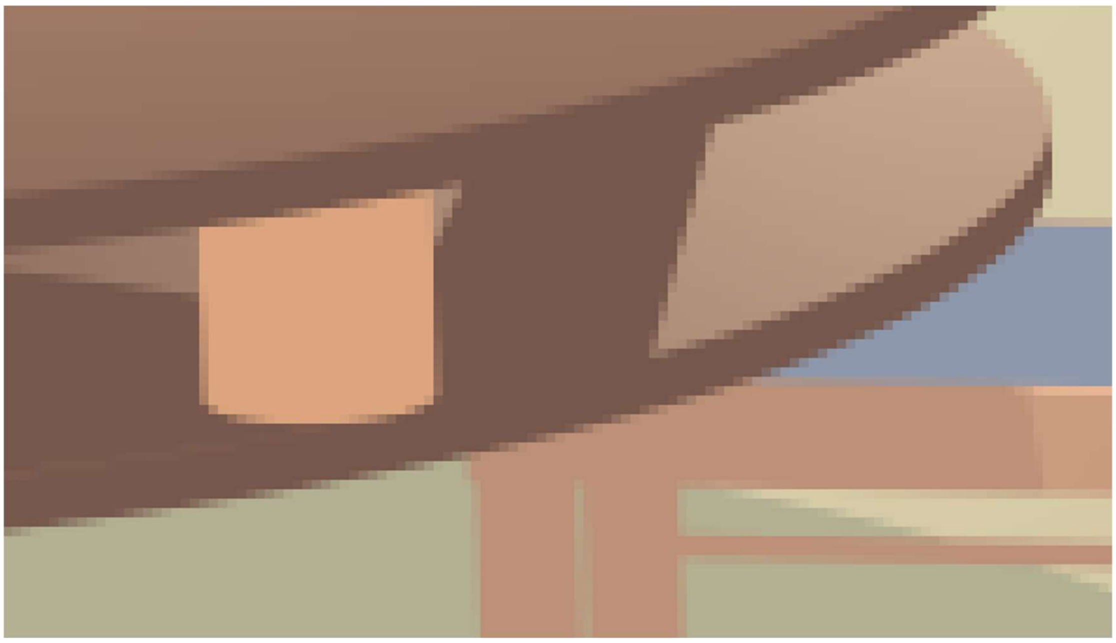


If we enlarge both the photos and focus on a small area, we can actually see the "mixed color" affect in the pixels:

No Anti-Aliasing used:



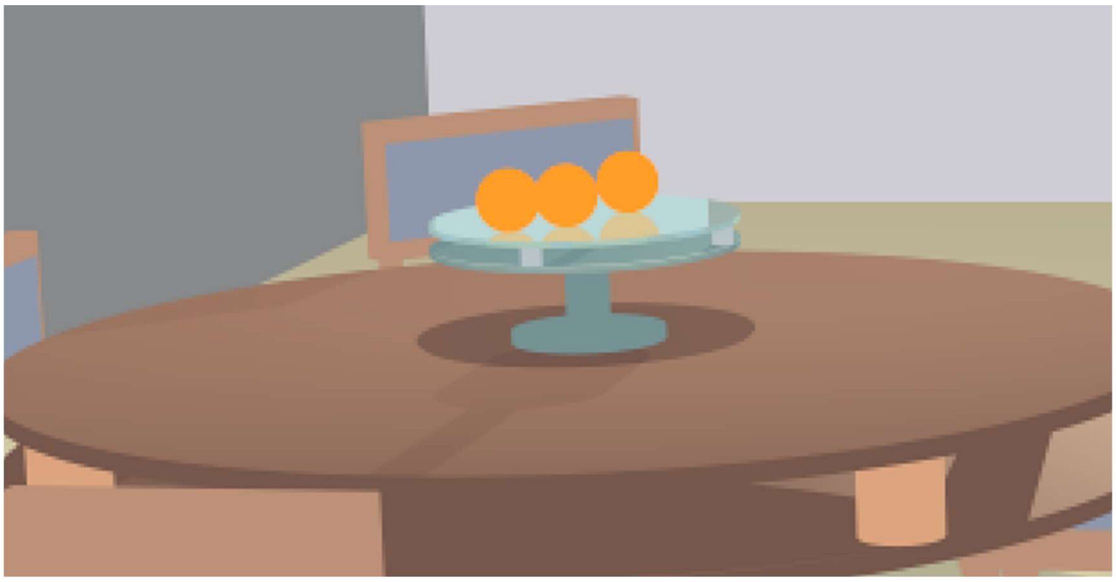
With the Anti-Aliasing:



Soft Shadow:

Another issue we noticed is that if we take a closer look at the shadows in the scene , we notice that they are very "hard", meaning, that we have an area that is shadowed completely and then right next to it we have an unshaded area with a very clear line between them, which does not resemble shadow in the real world were the shadow fades slowly until it is unshadow.

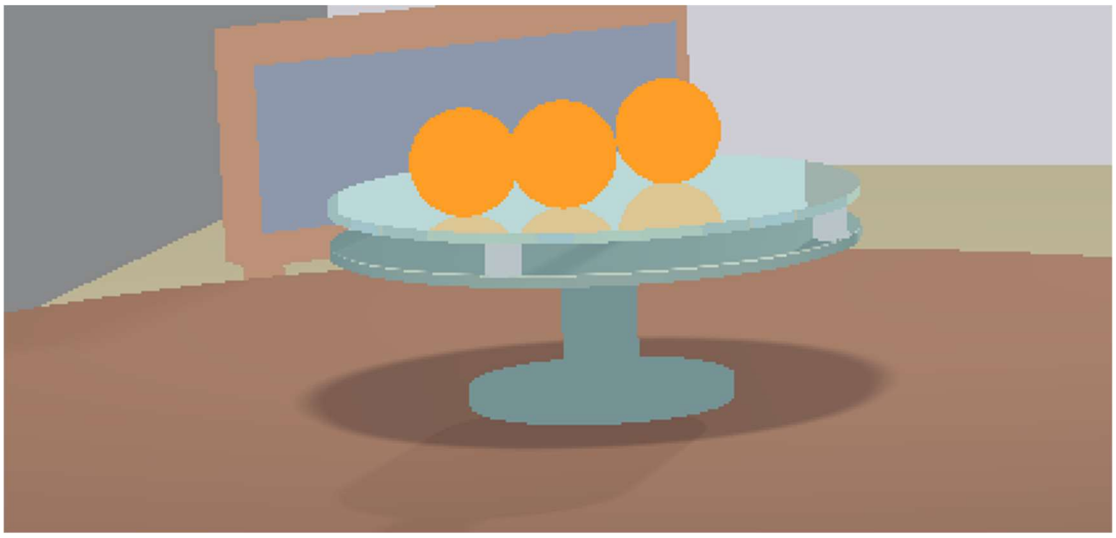
Here is a closer look at a problematic shadow in the scene:



we can see in the center of the table the platter creates a perfect circle of shade with no fading.

To solve this , instead of casting one ray from the intersection point towards the light source to see if the point is shaded we cast n*m rays towards a "radius" around the light source , we calculate the shade by combining the results of all the rays , the closer the rays are in direction to the ray from the intersection point to the light source all the rays will intersect the shadowing object and the point will be fully shaded, but further away from the center of the shadowing object towards the edges , some of the rays will miss the shadowing object adding some light to the color of the point , after adding all the ray colors we will get a lighter shade giving us the fading affect as we wanted.

Here is the same snapshot using the improvement:



We can tell that now that the edge of shade circle fades, and is not a full "hard" border.

We implemented this in our code by adding a function in the light sources classes:

{@credit to Yona Shmerla – we used a very similar implementation to his for this function}

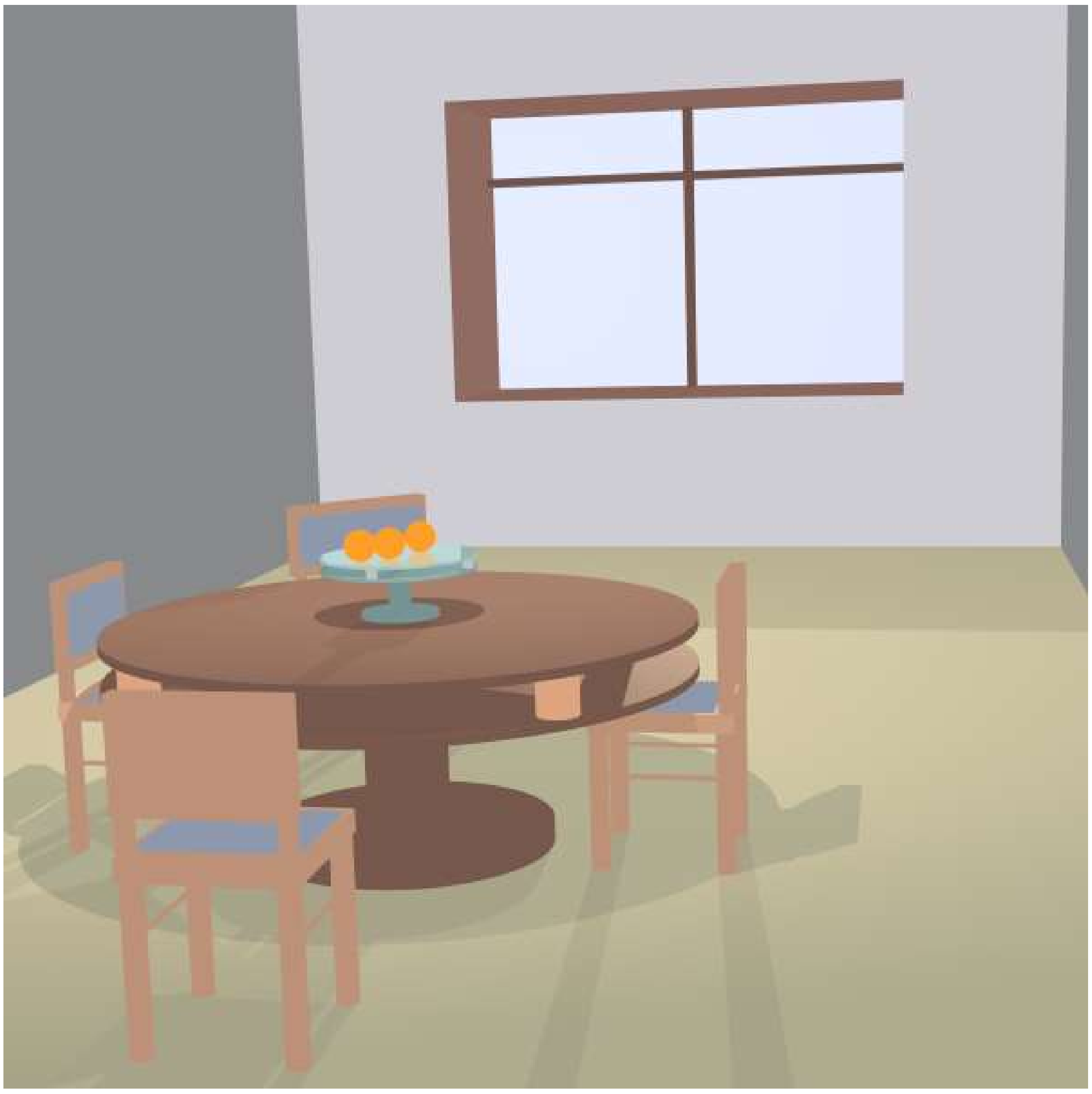
```
public List<Vector> getListL(Point p);
```

the function receives a point (the intersection point) and returns a list of rays constructed from the point to the above-mentioned radius round the light source. In the calcLocalEffects function we split the calculation to two options, if soft shadow is used, (again we added a boolean field in the camera class with a setter for easy use of functionality), we travers through the entire list to calculate the shade, if it is not used, the sole ray is used to calculate the shade as we explained earlier.

Results:

Finally, we present our full picture using both the improvements:

Anti-Aliasing – Random Method using 17X17 rays per pixel:



Soft Shadow: