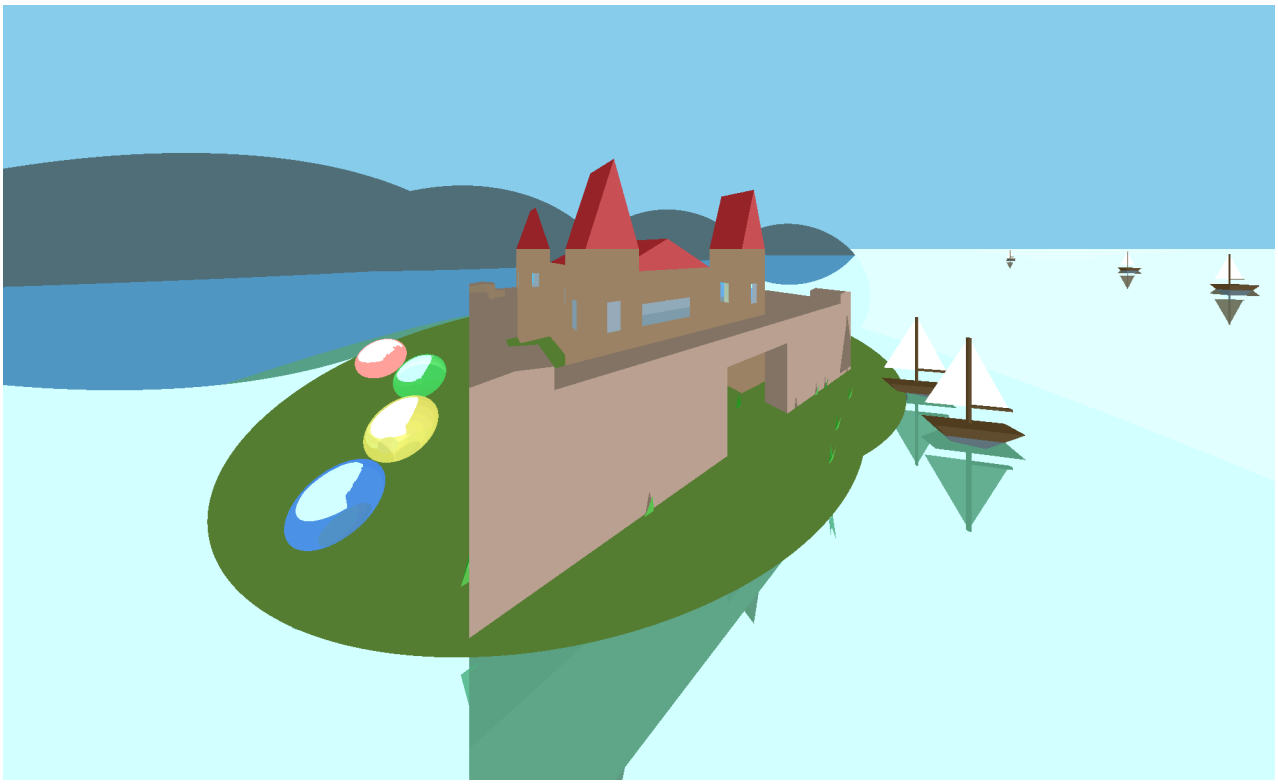
Mini Project In Introduction To Software Engineering - Project Report

Hadar Cohen

Einat Mazuz

Project Overview

Our project focused on developing a sophisticated scene by applying various software design patterns and programming principles. The primary objective was to efficiently and effectively create complex visual elements through the use of reusable object classes and corresponding test suites. This systematic approach allowed us to incrementally build a detailed and coherent scene.



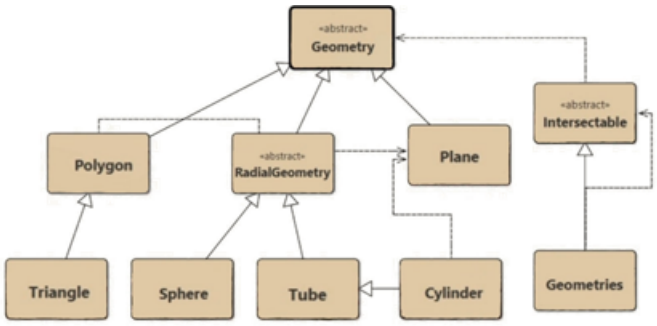
Key Programming Principles and Patterns :

Composite Pattern: We used the Composite Pattern to create complex shapes by extending simpler ones and utilizing interfaces. This approach allowed us to group various geometric forms into single entities, enabling easy manipulation of both individual components and the entire structure as a whole.

Strategy Pattern: In the implementation of the lighting interface, we applied the Strategy Pattern, allowing each light source to independently implement the interface's methods. This flexibility enabled seamless interchangeability of different lighting strategies, enhancing the overall versatility of the scene.

Builder Pattern: The Builder Pattern was employed in the construction of the camera, allowing us to assemble its attributes incrementally. This design ensured that the final object was both robust and adaptable, with a clear and organized construction process.

Iterator Pattern: To navigate through the geometries in images and objects, we utilized the Iterator Pattern. This provided an efficient way to traverse collections of elements, ensuring that every component was accessed and processed methodically.



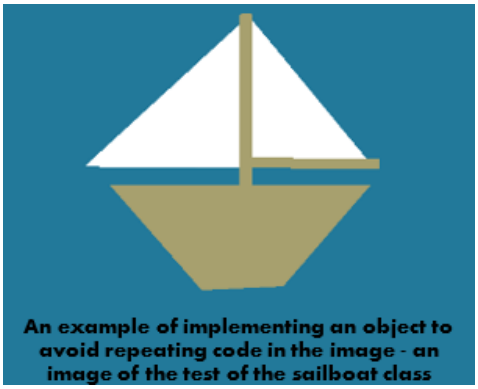
UML diagram of the geometries folder

Wrapper Pattern: We used the Wrapper Pattern in our custom color class to wrap Java's native color class. This allowed us to extend the functionality of color management while maintaining compatibility with Java's built-in features, providing a more versatile and powerful color manipulation tool.

Template Method Pattern: The Template Method Pattern was implemented in the Geometries class to standardize and simplify the interaction with various geometric bodies when constructing the scene. This ensured consistency in the underlying structure while allowing specific implementations to vary as needed.

Delegation Pattern: The Delegation Pattern was utilized in the Vector and Point classes to avoid unnecessary code duplication. By delegating specific operations to dedicated helper objects, we kept our codebase clean, modular, and easy to maintain.

Don't Repeat Yourself (DRY): In constructing the scene, we adhered to the DRY principle. For instance, in building objects containing multiple geometries, we created dedicated classes (with corresponding tests) to minimize repetitive code. This reduced redundancy and enhanced the maintainability and readability of our code.



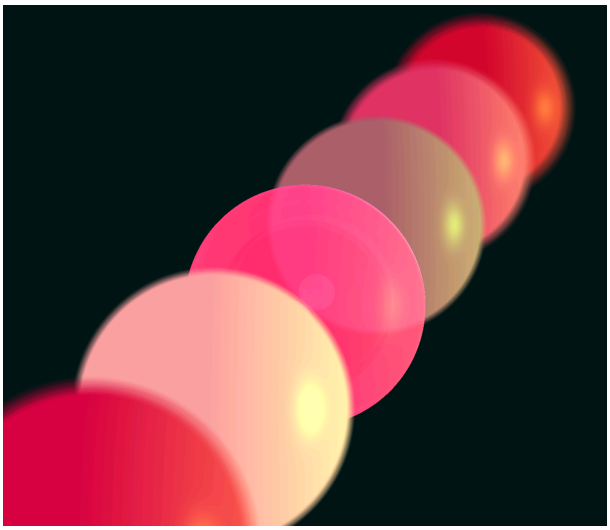
An example of implementing an object to avoid repeating code in the image - an image of the test of the sailboat class

By systematically applying these design patterns and principles, we were able to create a complex and visually appealing scene while maintaining high standards of code quality. This project demonstrates the effectiveness of disciplined software design in achieving both functional and aesthetic goals.

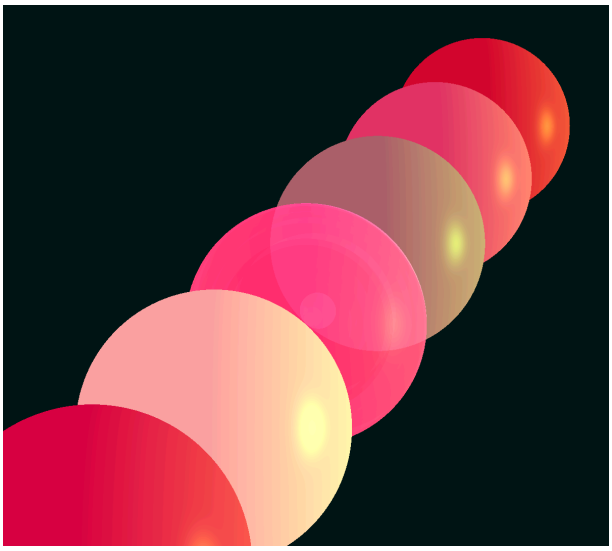
Mini Project 1: Depth of Field Enhancement

In this mini-project, we introduced the

Depth of Field effect, which allowed us to create a focused area within our scene, adding a layer of realism and visual interest. By simulating the way a camera lens focuses on specific objects while blurring others, we were able to direct the viewer's attention to key elements in the composition.



The effect was implemented by adjusting the focus distance and aperture size within our rendering algorithm, ensuring that only selected objects appeared sharp while the rest were gradually blurred.



implementation-

Added New Properties in Camera class:

apertureRadius: Introduced to define the radius of the camera's aperture. A larger radius results in a stronger DOF effect, causing more blur in out-of-focus areas.

focalLength: Added to specify the distance from the camera where objects appear sharp and in focus. This sets the focal point for the DOF effect.

DoFActive: A boolean property to activate or deactivate the DOF effect. When set to true, the camera will apply DOF calculations to create blurred areas for out-of-focus objects.

gridDensity: Added to control the density of the grid within the aperture area, affecting the number of rays used for DOF calculations. Higher values increase the accuracy but also the computation time.

DoFPoints: A list of points on the aperture plane used to calculate DOF. These points are randomly generated within the aperture to simulate light rays passing through different parts of the lens.

```
/** Aperture radius */
double apertureRadius = 0;

/** Focal length */
double focalLength = 0;

/** DoF active */
boolean DoFActive = false;

/**
 * Aperture area grid density
 */
int gridDensity = 1;

/**
 * DoF points on the aperture plane
 */
public List<Point> DoFPoints = null;

/**
 * Number of threads to use for rendering.
 */
```

Create New Method:

A method that generates a list of random points within a circular area on the aperture plane. These points simulate the different positions from which rays are cast, creating the DOF effect.

```java
/**
 * Generates a list of points randomly distributed within a circular area.
 *
 * @param gridDensity The number of points to generate.
 * @param radius      The radius of the circular area.
 * @param center      The center point of the circular area.
 * @param up          A vector representing the up direction for the circular
 *                    area.
 * @param right       A vector representing the right direction for the circular
 *                    area.
 * @return A list of points randomly distributed within the circular area.
 */
public static List<Point> generatePoints(int gridDensity, double radius, Point center, Vec
    List<Point> points = new ArrayList<>();

    for (int i = 0; i < gridDensity; i++) {
        double angle = 2 * Math.PI * Math.random();
        double r = radius * Math.sqrt(Math.random());
        double offsetX = r * Math.cos(angle);
        double offsetY = r * Math.sin(angle);

        Point point = center.add(right.scale(offsetX)).add(up.scale(offsetY));
        points.add(point);
    }
    return points;
}
```

Modified the renderImage Method:

```
if (DoFActive) {
    // Generate points on the aperture plane for depth of field effect
    this.DoFPoints = Camera.generatePoints(gridDensity,
                                           apertureRadius,
                                           location,
                                           vUp,
                                           vRight);

    // Handle the case where no points were generated
    if (this.DoFPoints == null || this.DoFPoints.isEmpty()) {
        // Fallback to a default value, using the camera location as the single point
        this.DoFPoints = List.of(location);
    }
}
```

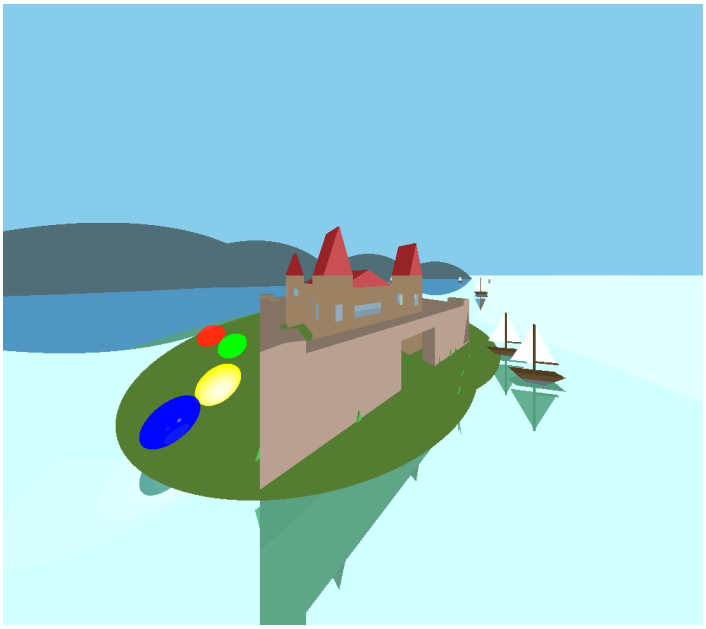
```
for (int i = 0; i < nY; ++i) {
    for (int j = 0; j < nX; ++j) {
        if (this.gridDensity != 1 && DoFActive) {
            // Calculate the focal point for depth of field
            var focalPoint = constructRay(nX, nY, j, i)
                             .getPoint(focalLength);

            // Write the pixel color using a bundle of rays for depth of field
            imageWriter.writePixel(j, i,
                rayTracer.computeFinalColor(
                    Ray.RayBundle(focalPoint, DoFPoints)
                ));
        } else {
            // Perform standard ray casting for the pixel
            castRay(nX, nY, j, i);
        }
    }
}
```

Summary:

- The shape is circular, resembling an open circular aperture.
- The distribution is random, but uniform across the area of the circle.
- The number of points is determined by the gridDensity.
- The size of the circle (aperture opening) is determined by the apertureRadius.
- The distance to the focal point is determined by the focalLength.

Without Dof



After applying DOF

Mini Project 2:
BVH (CBR) + MultiTheading

Utilizing Threads:

Problem-

Until now, we ran the images in a single process, which delayed the execution. We now want to improve the execution times.

Solution-

Therefore, we will separate the process during its execution into different and separate threads, where each thread will color different, non-overlapping pixels. To ensure that the threads truly color different pixels and do not work on the same pixel simultaneously, we created the Pixel class, which is responsible for selecting the next pixel to be colored. This ensures that the different threads work in parallel without overlapping.

```
if (threadsCount == 0) { // Single-threaded rendering
    for (int i = 0; i < nY; ++i) {
        for (int j = 0; j < nX; ++j) {
            if (this.gridDensity != 1 && DoFActive) {
                // Calculate the focal point for depth of field
                var focalPoint = constructRay(nX, nY, j, i).getPoint(focalLength);
                // Write the pixel color using a bundle of rays for depth of field
                imageWriter.writePixel(j, i, rayTracer.computeFinalColor(Ray.RayBundle(foc
            } else {
                // Perform standard ray casting for the pixel
                castRay(nX, nY, j, i);
            }
            // Update progress after processing each pixel
            pixelManager.pixelDone();
        }
    }
} else { // Multi-threaded rendering
    var threads = new LinkedList<Thread>(); // List to hold the threads
    while (threadsCount-- > 0) { // Create the required number of threads
        threads.add(new Thread(() -> {
            PixelManager.Pixel pixel; // Variable to hold the current pixel (row, col)
            // Loop until there are no more pixels to process
            while ((pixel = pixelManager.nextPixel()) != null) {
                if (this.gridDensity != 1 && DoFActive) {
                    // Calculate the focal point for depth of field
                    var focalPoint = constructRay(nX, nY, pixel.col(), pixel.row()).getPoi
                    // Write the pixel color using a bundle of rays for depth of field
                    imageWriter.writePixel(pixel.col(), pixel.row(), rayTracer.computeFina
                } else {
                    // Perform standard ray casting for the pixel
                    castRay(nX, nY, pixel.col(), pixel.row());
                }
                // Update progress after processing each pixel
                pixelManager.pixelDone();
```

Explenation:

1. **Single-threaded Mode:**
   - If threadsCount is 0, the entire process runs in a single thread, where each pixel is rendered one after the other.
2. **Multi-threaded Mode:**
   - If threadsCount is greater than 0, multiple threads are created to run in parallel. Each thread receives a pixel from the pixelManager, performs the necessary calculations (including depth of field if required), and colors it.
   - The threads work simultaneously, each rendering a different pixel, which speeds up the process. The program waits for all threads to finish before proceeding.

The goal is to improve rendering performance by distributing the workload across multiple threads running concurrently.

BVH Improvement:

The principle behind BVH (Bounding Volume Hierarchy) is to construct bounding boxes around objects and check for intersections with the objects inside the box. Each box is contained within a larger box, and the intersection calculations are performed recursively.

implementation:

We defined an inner class called BoundingBox in Intersectable , which stores the minimum and maximum coordinates of a box that is created around the object(s).

```
public class BoundingBox {
    private Point min = Point.NEGATIVE_INFINITY;
    private Point max = Point.POSITIVE_INFINITY;

    public BoundingBox(Point min, Point max) {
        this.min = min;
        this.max = max;
    }
```

We'll add a protected field of type BoundingBox to the Intersectable class, representing the bounding box of the object. In one of the constructors for the final objects (e.g., sphere and polygon), we'll calculate the size of the bounding box and store it in this field.

We'll also add a hasIntersection method to the Intersectable class to check if there are intersections between the ray and the bounding box. If there are no intersections with the bounding box, the method Intersectable.findGeoIntersections will not attempt to calculate intersections with the objects inside.

```java
/**
 * Checks if a ray intersects with this bounding box.
 *
 * @param ray The ray to test for intersection
 * @return true if the ray intersects the box, false otherwise
 */
public boolean hasIntersection(Ray ray) {

    // Check if Bounding Volume Hierarchy (BVH) is disabled or bounding box is null
    if (!BVH || boundingBox == null) {
        return true; // Proceed with further intersection checks
    }

    // Calculate intersection intervals for the X axis
    double xMin = (boundingBox.min.getX() - ray.getHead().getX())
                  / ray.getDirection().getX();
    double xMax = (boundingBox.max.getX() - ray.getHead().getX())
                  / ray.getDirection().getX();

    // Swap xMin and xMax if necessary
    if (xMin > xMax) {
        double temp = xMin;
        xMin = xMax;
        xMax = temp;
    }

    // Calculate intersection intervals for the Y axis
    double yMin = (boundingBox.min.getY() - ray.getHead().getY())
                  / ray.getDirection().getY();
    double yMax = (boundingBox.max.getY() - ray.getHead().getY())
                  / ray.getDirection().getY();

    // Swap yMin and yMax if necessary
    if (yMin > yMax) {
        double temp = yMin;
        yMin = yMax;
        yMax = temp;
    }
```

```
    // Check if the intervals on the X and Y axes overlap
    if ((xMin > yMax) || (yMin > xMax)) {
        return false; // No intersection
    }

    // Update xMin and xMax based on the Y axis intervals
    if (yMin > xMin) {
        xMin = yMin;
    }
    if (yMax < xMax) {
        xMax = yMax;
    }

    // Calculate intersection intervals for the Z axis
    double zMin = (boundingBox.min.getZ() - ray.getHead().getZ())
                  / ray.getDirection().getZ();
    double zMax = (boundingBox.max.getZ() - ray.getHead().getZ())
                  / ray.getDirection().getZ();

    // Swap zMin and zMax if necessary
    if (zMin > zMax) {
        double temp = zMin;
        zMin = zMax;
        zMax = temp;
    }

    // Check if the intervals on the X and Z axes overlap
    if ((xMin > zMax) || (zMin > xMax)) {
        return false; // No intersection
    }

    // Update xMin and xMax based on the Z axis intervals
    if (zMin > xMin) {
        xMin = zMin;
    }
    if (zMax < xMax) {
        xMax = zMax;
    }

    // Return true if there is an intersection in front of the ray's origin
    return xMax > 0; // Indicate that the ray intersects the bounding box
}
```

To unify the bounding boxes of the objects into groups, we add a public static method called buildBVH to the BoundingBox class. This method will receive a list of objects and return a new BVH structure .

buildBVH – Algorithm Description:

1. Separate the list of objects into finite and infinite objects.
2. Sort the finite objects according to their centers' coordinates along the X-axis (or any other parameter).
3. Divide the list of finite objects into two parts.
4. For each part, create a new BVH node, calculate the bounding box size for each part (recursively).
5. For infinite objects, create a new Geometries object and calculate the bounding box size.
6. Unite the bounding boxes of the finite and infinite objects into the Geometries object (which is the BVH) and return the result.

```java
/**
 * Builds a Bounding Volume Hierarchy (BVH) from a list of intersectable geometries.
 *
 * @param intersectableList The list of intersectable geometries to build the BVH from.
 * @return A list of intersectable geometries with the BVH structure applied.
 */
public static List<Intersectable> buildBVH(List<Intersectable> intersectableList) {

    // Base case: if the list has 1 or fewer geometries, return it as is
    if (intersectableList.size() <= 1) {
        return intersectableList;
    }

    // Extract infinite geometries into a separate list
    List<Intersectable> infiniteGeometries = new LinkedList<>();
    for (int i = 0; i < intersectableList.size(); i++) {
        var g = intersectableList.get(i);
        if (g.getBoundingBox() == null) {
            infiniteGeometries.add(g);
            intersectableList.remove(i);
            i--;
        }
    }

    // If there are no finite geometries left, return the infinite geometries
    if (intersectableList.isEmpty()) {
        return infiniteGeometries;
    }

    // Sort geometries based on their bounding box centroids along the X-axis
    intersectableList.sort(Comparator.comparingDouble(g ->
            g.getBoundingBox().getCenter().getX()));

    // Split the list into two halves
    int mid = intersectableList.size() / 2;
    List<Intersectable> leftGeometries =
        buildBVH(intersectableList.subList(0, mid));
    List<Intersectable> rightGeometries =
        buildBVH(intersectableList.subList(mid, intersectableList.size()));
```

```java
    // Create bounding boxes for the left and right geometries
    BoundingBox leftBox = getBoundingBox(leftGeometries);
    BoundingBox rightBox = getBoundingBox(rightGeometries);

    // Combine the left and right geometries into a single Geometries object
    Geometries combined = new Geometries();
    for (var g : leftGeometries) {
        if (g != null)
            combined.add(g);
    }
    for (var g : rightGeometries) {
        if (g != null)
            combined.add(g);
    }

    // Set the bounding box for the combined geometries
    if (leftBox != null && rightBox != null) {
        combined.boundingBox = leftBox.union(rightBox);
    } else if (leftBox != null) {
        combined.boundingBox = leftBox;
    } else if (rightBox != null) {
        combined.boundingBox = rightBox;
    }

    // Combine the infinite geometries with the finite ones and return the result
    List<Intersectable> result = new LinkedList<>(infiniteGeometries);
    result.add(combined);
    return result;
}
```

We will add a short method in the Geometries class to rebuild all the objects into a BVH structure:

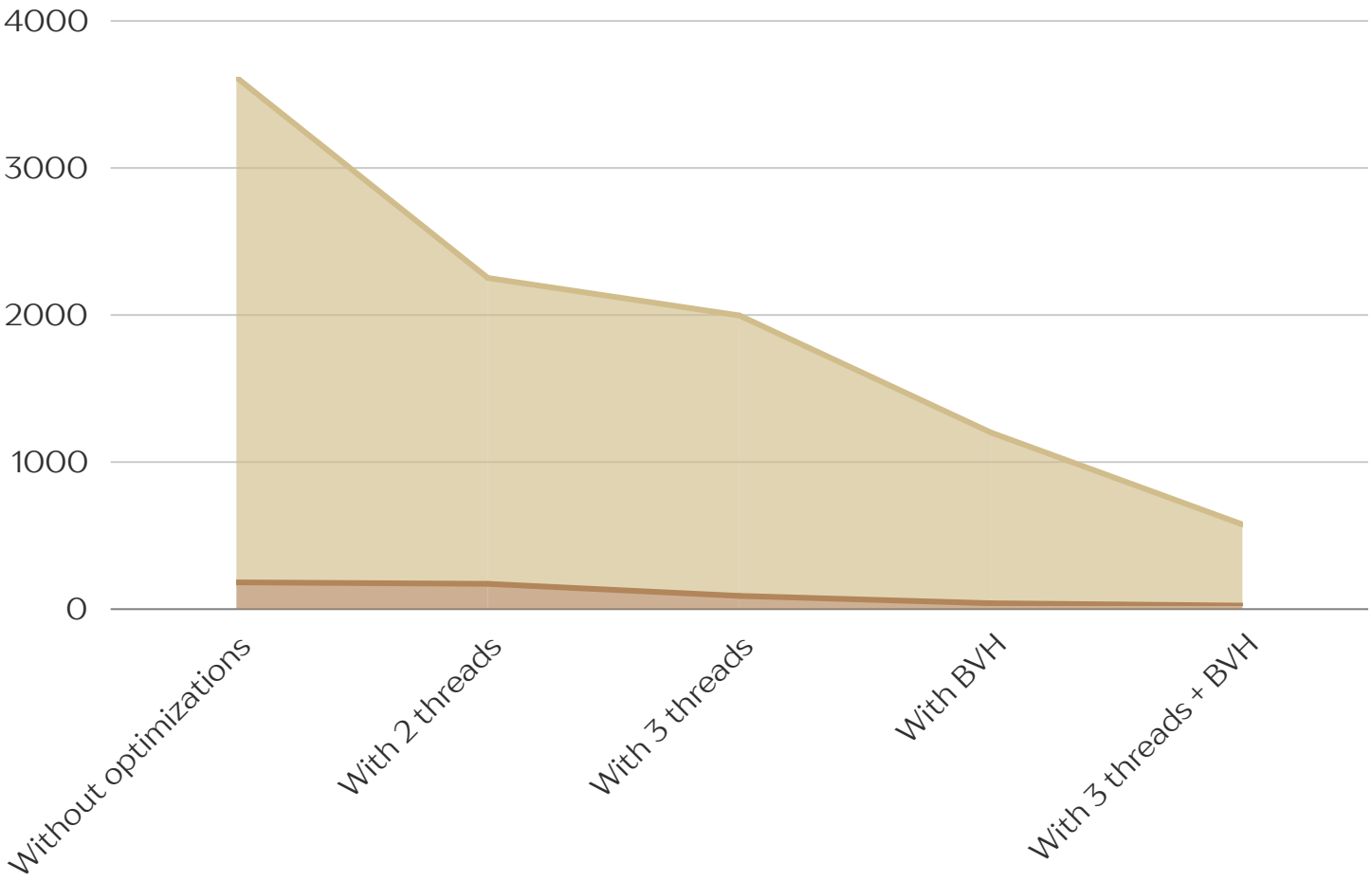
```java
/**
 * Converts the current list of geometries into a BVH structure.
 */
public void makeBVH() {
    // Build the BVH from the current geometries
    List<Intersectable> intersectables = BVHBuilder.buildBVH(geometries);

    // Clear the existing geometries list
    geometries.clear();

    // Add the BVH-structured geometries back to the list
    geometries.addAll(intersectables);
}
```

After building a scene in the tests and adding all the objects, we can reduce them to a BVH structure, thus saving time during rendering calculations. When we want to use the BVH, we will add:

```java
// After building the scene and adding all objects, optimize with BVH
scene.geometries.makeBVH(); // Should improve performance during rendering
```



Graph Explanation:

- **The execution times of our main image (on a specific computer) using the optimizations we implemented:**
- The execution time of the image without any optimization is 3 minutes and 2 seconds.
- The execution time of the image using 2 threads is 34 minutes and 40 seconds.
- The execution time of the image using 3 threads is 1 minute and 30 seconds.
- The execution time of the image using BVH only is 40 seconds.
- The execution time of the image using 3 threads along with BVH is 24 seconds.
- With DOF:
- The execution time of the image without any optimization is 57 minutes and 14 seconds.
- The execution time of the image using 2 threads is 34 minutes and 18 seconds.
- The execution time of the image using 3 threads is 31 minute and 47 seconds.
- The execution time of the image using BVH only is 19 minute and 21 seconds.
- The execution time of the image using 3 threads along with BVH is 9 minute and 12 seconds.

Bonus Features:

- **Exercise 3**: Implementation and testing of ray-polygon intersection.
- **Exercise 5**: Building a scene from an XML or JSON file.
- **Exercise 7**: Create an image that includes 10 or more geometries, demonstrating all the features and effects applied.

Conclusion

Through the careful application of OOP principles, parallel processing, and advanced rendering techniques, our project successfully created a visually complex and efficient scene. The enhancements implemented in our mini-projects not only improved the visual quality of the scene but also optimized performance, making our approach both effective and scalable.