Raytracer (18 Punkte)

In this project your task is to write a raytracer. The goal of the project is to gain hands-on experience in object-oriented design and programming. Clone the project into any virtual machine folder for the lecture:

```
git clone ssh://git@git.prog2.de:2222/project5/$NAME.git project5
```

where you have to replace $NAME with your CMS username.
You can edit the project using *VSCode*.
Perform the following steps in VSCode to open the project:

1. Select *File* → *Open Folder...*.
2. In the file explorer select `project5` (or the folder where you have cloned it)
3. When asked, select "Yes, I trust the authors".

For testing, we use a test framework for Java called *JUnit*. It runs regression tests on your implementation and reports the result of each. Clicking on the button with an Erlenmeyer flask on it (the lowest in the column of buttons on the top-left) opens the testing user interface. With the double arrow on the top you can start all tests and the arrow with the bug lets you debug them.

1 Tasks

The tasks to be completed are listed below. They are each provided with references to the relevant sections with explanations. Section 2 provides the basics.

Calculate Hits (6 Points) Implement the `Plane` and `Sphere` (3.2) classes. You can use the existing implementation in the `Triangle` class as a guide. Also implement the methods of the static factory for objects `GeomFactory`. Implement the `hashCode()` and `equals(Object)` methods for the `Plane` and `Sphere` classes.

Shader (4 points) Implement the `shade()` method of the `CheckerBoard` and `Phong` (4.1) classes. You can use the existing implementation in the `SingleColor` class as a guide. For shaders there is a static factory `ShaderFactory`.

Read objects (3 points) Implement the `read()` method of the `OBJReader` class to read OBJ files (5.1). Also make sure to scale and move the read object correctly.

Acceleration structure (5 points) Implement the `BVH` class of the bounding"=volume"=hierarchy"=acceleration structure (4.3).

Hints

- You can start Raytracer as a normal program using this ▷ on the top-right corner of the VSCode interface. The `Main` file is located in the package `raytracer.core.def`. The output is an image containing (at first) only a yellow triangle.
- In `Main.main()` are several boolean variables that you can use to turn on other objects and effects in the scene once you have implemented the appropriate part. For example after implementing the `Sphere` class and setting `implementedSphere` to `true` two spheres will appear. This way you can easily survey their progress.
- In attachment 1 there is a reference image generated after implementing all relevant parts.

- Always use the constant `EPS` of the class `Constants` for comparisons with the number 0. A comparison of the form $i \leq 0$ thus becomes the comparison $i <$ EPS. For example, if you test a number to 0, you can use the `isZero` method of this class.
- Files to be edited are: `geom/GeomFactory.java`, `shade/ShaderFactory.java`, `core/OBJReader.java` and `core/def/BVH.java`. All other *existing* files will be overwritten. (Nevertheless you are encouraged to edit the `Main` as stated above; this has no impact on the test cases.) Additionally you can and should add new classes in new files. Don't forget to add new Java files in git too!
- In the JavaDoc you can find more detailed specifications directly at the methods to be implemented, especially about which exceptions should be thrown. The first specified exception whose condition is true should always be thrown.
- Most of the vector operations and operations on rays are already implemented. Take a look at the existing classes in the package `math` before you implement something that might already be there.
- Whenever you encounter problems, make use of the debugger! Clicking on the ▷ with the bug on the left hand side in VSCode opens the debugger interface.

2 Basics

The basic idea is to reproduce natural visual impressions. In reality, light sources, like the sun, send light rays into space. These light rays are changed in many ways. They are scattered by matte surfaces (e.g. walls), reflected by smooth surfaces (e.g. mirrors) or refracted by transparent objects (e.g. water). Some light rays strike the eye of an observer and are finally perceived as an image.

2.1 Modell

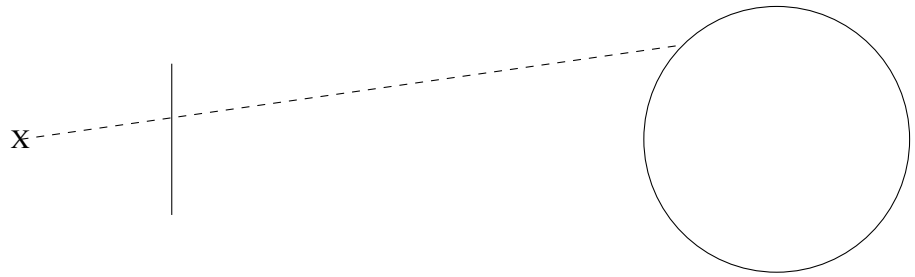

Figure 1: A ray of vision starting at the camera travels through the projection plane and hits a sphere.

To reproduce this exactly is extremely complex, because most of the light rays do not reach the observer. Therefore, an approach is taken in the opposite direction: First, the observer is modeled as a camera with a projection plane on which the resulting image is drawn. Now "viewing rays" are sent from the camera into the scene. These may hit objects there. Their color is calculated at the hit points and entered in the resulting image (the projection plane). This is illustrated in Figure 1. In order to reproduce effects such as illumination or reflection, further rays are shot into the scene from the hit points (e.g. in the direction of light sources for shadow calculation).

2.2 Linear Algebra

In this section, the basics of linear algebra necessary for the project are briefly covered. These are then applied to the hit and color calculations.

Points A point is a location in a coordinate system relative to its origin. Lowercase letters (a) are used as names in the following. In a three-dimensional coordinate system, a point is uniquely described by three numbers.

Vectors Vectors specify a displacement in a three-dimensional space by means of three numbers. Vectors are written as lowercase letters with an arrow above ($\vec{a}$). The most important operations with vectors are listed below:

- The element-wise sum of two vectors gives a combined displacement.
- The element-wise difference of two points $a - b$ results in a vector that gives the direct path from b to a.
- The element-wise multiplication of two vectors is represented by the operator symbol $*$.
- A vector can be multiplied element-wise by a number k to compress ($|k| < 1$) or stretch ($|k| > 1$) it. If the factor is negative, the resulting vector points in the opposite direction. If two vectors can be converted into each other with such a scalar multiplication, they are called linearly dependent.
- A vector elementwise added to a point yields a point shifted by this vector.
- The length of a vector is calculated according to Pythagoras' theorem:

  $$|\vec{a}| = \sqrt{a_1^2 + a_2^2 + a_3^2} = \sqrt{\vec{a}^2}$$

  The variant of the calculation on the right uses the scalar product, see section 2.3. Vectors of length 1 are called unit vectors.

2.3 Vector operations

Scalar product The scalar product (also called "dot product", "inner product" or "direct product") is a union of two vectors and gives as result a number which indicates which angle they enclose:

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos \alpha$$

Since we are working with a three-dimensional Cartesian coordinate system, the scalar product is calculated as the sum of the element-wise products of the vector entries:

$$\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + a_3 b_3$$

Sometimes, analogous to scalar multiplication, the dot is omitted. Accordingly, $\vec{a}^2$ is written for the scalar product of a vector with itself. Dividing the scalar product by the product of the length of the vectors results in the following special values (according to the cosine):

- 1: The two vectors are parallel and point in the same direction.
- -1: The two vectors are parallel and point in opposite directions.
- 0: The two vectors are perpendicular to each other.

Values in between indicate how much the two vectors point in the same (> 0) or opposite (< 0) direction.

Cross product In three-dimensional space, the cross product (also called "vector product") calculates a third vector from two vectors that is perpendicular to both vectors (scalar product with the two vectors is 0). Since the length of the resulting vector depends on the angle enclosed by the two given vectors, the result usually still needs to be normalized. It is calculated as follows:

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix}$$

2.4 Coordinate systems

World coordinates We use a left-handed coordinate system in the project. That is, the "X" direction points to the right, the "Y" direction points up, and the "Z" direction points away from the viewer. This can be seen vividly – along with an explanation of the term "left-handed" – in Figure 2.

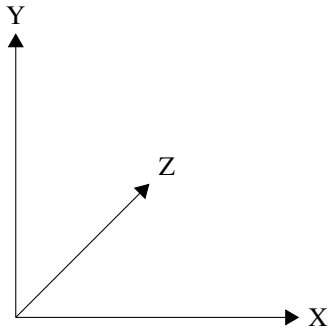

Figure 2: A left-handed coordinate system: the thumb of the left hand points in the "X"=direction, the index finger in the "Y"=direction, and the middle finger in the "Z"=direction.

Texture coordinates While the objects are in a three-dimensional scene we consider only two-dimensional textures. If you put a texture on an object, you have to calculate the texture coordinates for the intersection point. These are usually denoted by U and V. A `Hit` returns these coordinates using `getUV()` (see class `Triangle`).

3 Modeling

3.1 Camera

A camera observes the scene. From it, beams are shot into the scene. At the point of impact of each ray, a color is determined depending on the object hit, its surface, the lighting conditions there, etc. The color of the ray is then determined by the camera. For this purpose, additional beams are often shot into the scene, e.g. to determine whether a light source reaches the point of impact of the first beam. The color determined in this way is entered into the resulting image. We use a ready-made perspective camera in the project.

3.2 Objects

Planes (3.2) and spheres (3.2) are popular as geometric primitives, since the calculations of whether they are visible at a point are very simple. Triangles are also frequently offered, since they are usually the basis of polygon models. The calculation here is similar to that of planes, but requires further checks.
Objects in the scene have two important methods `hitTest()` and `shade()`. `hitTest()` is used to determine whether a ray hits the object. Another important result of this method is at what distance from the camera the object is hit. The object that is hit closest to the camera is then used to calculate the color of the pixel on the projection plane. This is done using `shade()`, which is discussed in section 4.1.

Rays We first consider rays, since we will intersect them with objects in the scene later on to determine points of impact. A ray is uniquely described by a point p_s, which specifies the starting point of the ray, and a vector $\vec{v}_s$, which specifies the direction of the ray. All points x that satisfy the following equation lie on the ray:

$$x = p_s + \lambda \vec{v}_s, \lambda \geq 0$$

λ is the distance from the starting point p_s. For $\vec{v}_s$ a unit vector is used.

Plane A plane is intuitively uniquely described by three points that do not lie on a straight line: For example, you can balance a book (the plane) without wiggling (uniquely) on three fingertips (the points).
However, this shape is not suitable for cutting with a ray. For this purpose the Hesse normal form is used. It consists only of a vector $\vec{n}_e$ perpendicular to the plane and the distance d_e of the plane from the origin. Every point x that satisfies the following equation lies in the plane:

$$x\vec{n}_e - d = 0$$

To obtain the Hesse normal form from three points which do not lie on a straight line, proceed as follows: From the three points first an arbitrary point p_e is chosen and two vectors $\vec{u}_e$ and $\vec{v}_e$ (path from the chosen point to the

two other points) are calculated. Since the three points do not lie on a straight line, the vectors $\vec{u}_e$ and $\vec{v}_e$ are not parallel to each other and in particular $|\vec{u}_e| \neq 0 \neq |\vec{v}_e|$ is also valid. This representation also spans the plane:

$$x = p_e + \lambda\vec{u}_e + \mu\vec{v}_e, \lambda, \mu \in \mathbb{R}$$

Now, using the cross product, the normal vector is calculated from the two vectors:

$$\vec{n}'_e = \vec{u}_e \times \vec{v}_e$$

The result of the cross product is usually not a unit vector, so it still needs to be normalized:

$$\vec{n}_e = \vec{n}'_e \frac{1}{|\vec{n}'_e|}$$

Now only the distance d_e of the plane from the origin has to be calculated. Since p_e is in the plane, this is done by substituting in and transforming the Hesse normal form,[1].

$$p_e\vec{n}_e = |p_e||\vec{n}_e|\cos\alpha = |p_e|\cos\alpha = |p_e|\frac{d}{|p_e|} = d$$

where α is the angle enclosed by $\vec{n}_e$ and p_e (interpreted as a vector from the origin). The elimination of $\cos\alpha$ comes from the fact that d is the length of the adjacent and $|p_e|$ is the length of the hypotenuse of the right triangle spanned by p_e and the extension of $\vec{n}_e$ to the plane.
The intersection of a ray with a plane is now calculated as follows ($\vec{v}_s$ and $\vec{n}_e$ are normalized):

$$\begin{array}{rcll} x & = & \lambda\vec{v}_s + p_s & \texttt{(ray)} \\ 0 & = & x\vec{n}_e - d_e & \texttt{(plane)} \\ 0 & = & (\lambda\vec{v}_s + p_s)\vec{n}_e - d & \texttt{(insert)} \\ 0 & = & \lambda\vec{v}_s\vec{n}_e + p_s\vec{n}_e - d & \\ \lambda\vec{v}_s\vec{n}_e & = & d - p_s\vec{n}_e & \\ \lambda & = & \frac{d-p_s\vec{n}_e}{\vec{v}_s\vec{n}_e}, \vec{v}_s\vec{n}_e \neq 0 \wedge \lambda \geq 0 & \end{array}$$

If the denominator is 0, the ray is parallel to the plane and thus the plane is not hit. Otherwise, λ indicates the distance from the starting point of the ray at which the plane is hit. Furthermore, λ must not be negative, otherwise the intersection point is behind the starting point of the ray.
Hint: Use the already implemented `geom.Util.computePlaneUV()` method to complete `getUV()` (in the `Plane` and `Sphere` classes). The parameters of `computePlaneUV()` are the normal and the receptor point of the plane and the point of impact.
In the `hit()` method, parameters `tmin` and `tmax` are passed. These describe a range as distance from the start point of the ray in which a hit is "good".

Spheres All points x that are at a distance r_k from the center c_k of a sphere form the surface of a sphere:

$$|x - c_k| = r_k$$

Since r_k is non-negative, one can save the square root calculation in the length determination of the vector by squaring the equation:

$$(x - c_k)^2 = r_k^2$$

After substituting the equation for one ray one obtains:

$$\begin{array}{rcl} r_k^2 & = & (\lambda\vec{v}_s + p_s - c_k)^2 \\ 0 & = & \lambda^2\vec{v}_s^2 + 2\lambda\vec{v}_s(p_s - c_k) + (p_s - c_k)^2 - r_k^2 \end{array}$$

λ can now be determined using the usual solution formula for mixed quadratic equations:

$$\begin{array}{rcll} a & = & v_s^2 = 1 & \texttt{(unit vector)} \\ b & = & 2\vec{v}_s(p_s - c_k) & \\ c & = & (p_s - c_k)^2 - r_k^2 & \\ \lambda_{1,2} & = & \frac{-b \pm \sqrt{b^2-4c}}{2} & \end{array}$$

[1] You don't have to understand the following derivations, but you should want to

If the discriminant (value under the root) is non-negative, the ray and the sphere intersect. Further, the smaller solution for λ must not be negative, otherwise the intersection point is again behind the starting point of the ray. The larger solution is always to be ignored, since it corresponds to the meeting of the sphere from the inside on the back side.
Hint: Use the already implemented `geom.Util.computeSphereUV()` method to complete `getUV()`. The parameter of `computeUV()` is the vector from the center of the sphere to the point of impact.
In the `hitTest()` method parameters `tmin` and `tmax` are passed. These describe a range as distance from the start point of the ray in which a hit is "good".

Bounding Box A bounding box represents a virtual cuboid in space that encloses objects. In the base case, exactly one object is enclosed, e.g., the bounding box of a sphere has as side lengths exactly the diameter of the sphere. For the plane, there is a special infinite bounding box. In order to quickly calculate sections with a bounding box, they are always aligned with the axes of the coordinate system. The use of bounding boxes to accelerate image computation is discussed in the section 4.3.

4 Light sources

Different types of light sources can be used to illuminate the scene, each approximating different aspects of real light sources.
Point light sources emit light uniformly in all directions from a single point in space. They are suitable for simulating, for example, light bulbs. Lighting is further described in section 4.2.
Ambient light sources uniformly illuminate every part of an object. They are a crude simulation of scattered light. Thus, a room into which light falls only through a window is not illuminated only at the point where the light strikes. Due to light scattering, other parts are slightly illuminated. To simulate this effect realistically is very costly, so sometimes one is satisfied with the simple trick of an ambient light source. Since there can be different ambient light conditions at different parts of a scene, this aspect is not implemented as a light source, but as part of a shader, here Phong (4.1).
In the implementation the light sources are available through the `Scene` object.

4.1 Shader

Shaders give the surfaces of objects their appearance. We use three typical shaders in this project.

Simple color The simplest shader always provides a constant color regardless of other conditions, such as light sources. This can then be used as the basis for other shaders. It is also good for testing and debugging because of its simplicity.

Checkerboard A popular shader is the checkerboard. Here, two other shaders are displayed alternately on a grid.
Which shader to use follows from the given texture coordinates u and v as well as a scaling factor s, which indicates the size of the grid:

$$x = \lfloor u/s \rfloor + \lfloor v/s \rfloor$$

If x is even, the first shader is used, otherwise the second.

Phong The Phong shader, named after its inventor, gives a surface a plastic-like appearance. Its result is calculated from three parts:

$$I_{\texttt{Phong}} = I_{\texttt{ambient}} + I_{\texttt{diffuse}} + I_{\texttt{specular}}$$

Ambient light ($I_{\texttt{ambient}}$) For illumination, a fixed color is added to the result color as the contribution of ambient light.

Diffuse Scatter ($I_{\texttt{diffuse}}$) This simulates scattering from a rough surface. The result depends on the color of the light source c_l, the color of the underlying shader $c_{\texttt{sub}}$, the angle of incidence (the steeper the brighter) of the beam on the surface, and a material constant $k_{\texttt{diffuse}}$. This amount is calculated for each light source reaching this spot. This means that the beam path to the respective light source is not obscured (4.2) and the light hits the surface from the front (scalar product of normal of the surface $\vec{n}$ and vector to the light source $\vec{v}$ greater than 0). Both vectors must be normalized. The formula for the calculation of diffuse scattering is

$$I_{\texttt{diffuse}} = \sum_{l \in L} (c_l * c_{\texttt{sub}}) k_{\texttt{diffuse}} max(0, \vec{n}\vec{v}),$$

where the $*$ stands for the component-wise product of the four-dimensional color vectors and L is the set of *relevant* light sources – see 4.2.

Mirroring ($I_{\texttt{specular}}$) This aspect simulates specular reflection of light. The result depends on the color of the light source c_l and the angle between the reflection vector $\vec{r}$ and the vector $\vec{v}$ from the point of incidence to the light source. The reflection vector is obtained by mirroring the direction vector of the light beam on the normal of the surface (there is a pre-implemented method for this). Further important are the material constants of the reflection factor ($k_{\texttt{specular}}$) and the gloss factor (n). The gloss factor ("shininess") describes the smoothness of the surface. The smaller the value, the rougher the surface appears, $n = \infty$ corresponds to a perfect mirror.

$$I_{\texttt{specular}} = \sum_{l \in L} c_l k_{\texttt{specular}} max(0, \vec{r}\vec{v})^n$$

4.2 Illumination

Before a light source is used to illuminate an object, it must be tested whether the light from the light source can reach the object at all. For this purpose, another ray – a shadow ray – is shot from the calculated point of incidence of the visual ray in the direction of the light source. If another object is hit on the way, the light source is covered by it and thus does not contribute to the illumination of the object. Without light, everything is dark, so an empty set L leads to the color black for $I_{\texttt{diffuse}}$ and $I_{\texttt{specular}}$.

4.3 Acceleration structure

Motivation Each ray shot into the scene must be tested for intersection with every object in the scene. For complex scenes, this is very slow. You can speed this up tremendously with a simple consideration: You put several objects in a box surrounding them. If a ray does not hit the box, the test of the individual objects in the box is not necessary. Such a box is called a *bounding box*.
As an illustration, consider Figure 3. To keep it simple, we use two-dimensional example, but the process works analogously for three (or even more) dimensions. The dashed lines define the borders of the corresponding bounding boxes and they form a hierarchy:

- the black bounding box contains the blue bounding box and the red bounding box
- the blue bounding box contains the orange and the green bounding box
- and the green, orange and magenta bounding boxes just contain one object, respectively

If there is a ray shot into the scene that does not hit the blue box, we already know that neither the parallelogram nor the triangle are hit, and thus there is no need to test those individually for hits.
This way we can now build a hierarchy of bounding boxes: Initially, we put all objects in one box (the black box in the example). Then we can recursively build two sub-boxes for every box and distribute the objects accordingly, as long as there are objects to distribute. This data structure is called Bounding Volume Hierarchy (BVH). Due to its hierarchical nature the BVH allows to test only a logarithmic number of boxes/objects for hits instead of a linear number (i.e. every object once). Note that the sub-boxes do not necessarily fully cover their parent box and that they may intersect with each other.
There are many different possible strategies for distributing the objects into sub-boxes. They may lead to different BVHs and possibly different performance on different scenes. The next paragraph describes the specific approach used in this project.

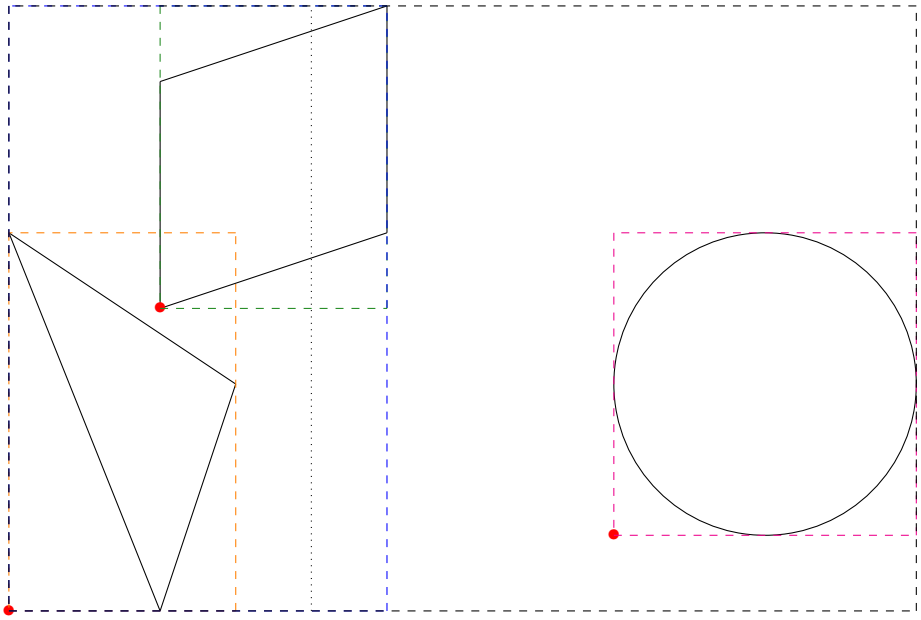

Figure 3: Minimal axis-aligned bounding boxes for two-dimensional shapes

Implementation A bounding box is the *minimal axis-aligned* box that surrounds its members. The *minimum point* / *maximum point* of a bounding box (`getMin()` / `getMax()` in BBox) is the point with minimum / maximum X, Y and Z coordinate. For the two-dimensional case the minimum points are marked with a red dot in Figure 3; the maximum points are the corners on the other side, respectively. The minimum point and the maximum point together describe the dimensions of a bounding box. The bounding box itself is already implemented (in the class `BBox`). For any object A we also call the minimum point of its minimal axis-aligned bounding box the *minimum point* of A. Thus the red dots in Figure 3 are also the respective minimum points of the triangle, the parallelogram and the circle.
To build the BVH, we start with a bounding box that contains all the objects. In the method `buildBVH()` the sub-boxes are built recursively. To this end, the following auxiliary methods are important:

- `add`: Add an object or a bounding box to the BVH by adjusting the corresponding bounding box as well as the list of members.
- `calculateMaxOfMinPoints`: This method should compute the component-wise (i.e. for each coordinate separately) maximum of the minimum points of all contained objects.
- `calculateSplitDimension`: This method should determine into which dimension the box should be split. Here we want to use the dimension where the difference between the minimum point of the box [2] and the component-wise maximum of the minimum points (`calculateMaxOfMinPoints`) is the largest.
- `distributeObjects`: If i is the dimension determined by `calculateSplitDimension`, then the box is split in dimension i at the midpoint between the i-th component of the minimum point of the box and the i-th component of the maximum of the minimum points of its members. Each object shall be distributed accordingly into one of the two sub-boxes. The dotted line in Figure 3 shows the split of the black bounding box according to which the objects should be distributed into the sub-boxes: All objects *whose minimum point* is left to the line go into the blue bounding box and all the others are put into the red bounding box.

Make sure that your algorithm is terminated: You may have to add an additional check for further subdivisions to prevent infinite recursion.
For small numbers of objects the computational overhead of the BVH is greater than its performance gain thus we stop the splitting of bounding boxes as soon as only 4 or less objects are contained in it. After the build, every BVH object should either contain exactly two BVHs or 4 or less primitive objects.

[2] which is equal to the component-wise minimum of the minimum points of all contained objects (by construction of the bounding box)

The method `hit` should make use of the structure as it was outlined in Paragraph 4.3. If a BVH is hit by a ray, the sub-boxes shall be recursively checked for hits. Only the first hit from the ray source with a primitive object should be returned – as in the case without BVH.

5 Models

It is impractical to hard-code a scene in the ray tracer source code. Therefore, many interchange file formats have been invented. Our raytracer should be able to read one of them (in excerpts).

5.1 Wavefront OBJ files

```
# comment
v 0.0 0.0 0.0
v 1.0 0.0 0.0
v 0.0 1.0 0.0
v 1.0 1.0 0.0
f 1 2 3
f 3 2 4
```

Figure 4: An OBJ file describing a square consisting of two triangles.

Wavefront OBJ files describe polygon models. We restrict ourselves to models consisting of triangles.
OBJ files consist of a list of definitions for vertices, sides, normals, texture coordinates, and so on. Each definition occupies one line and starts with a word describing the type of definition. An example of an OBJ file describing a square consisting of two triangles is shown in Figure 4. We consider only vertices and sides, whose structure is described below. All other definitions and comment lines (starting with a #) are to be ignored.

Vertices Definitions for vertices start with the word `v`, followed by three floating point numbers specifying the X, Y, and Z"=coordinates of the vertex. The vertices are entered into a list starting at index 1 for later use in pages.

Faces Definitions for faces start with the word `f`. This is followed by at least three indexes into the list of vertices, which represent the vertices of the polygon. To use a vertex in a face, it must have been defined before the face. We restrict ourselves in the project to triangles, i.e. only faces with three indices are used. It is guaranteed that all vertices to which a face refers have already been defined in the file before the face was defined.

Note Use `java.util.Scanner` to read the file. After a scanner is created, call the `useLocale()` method with the `Locale.ENGLISH` argument, otherwise points in the file to be read in will not be recognized as decimal separators.

1 Reference image

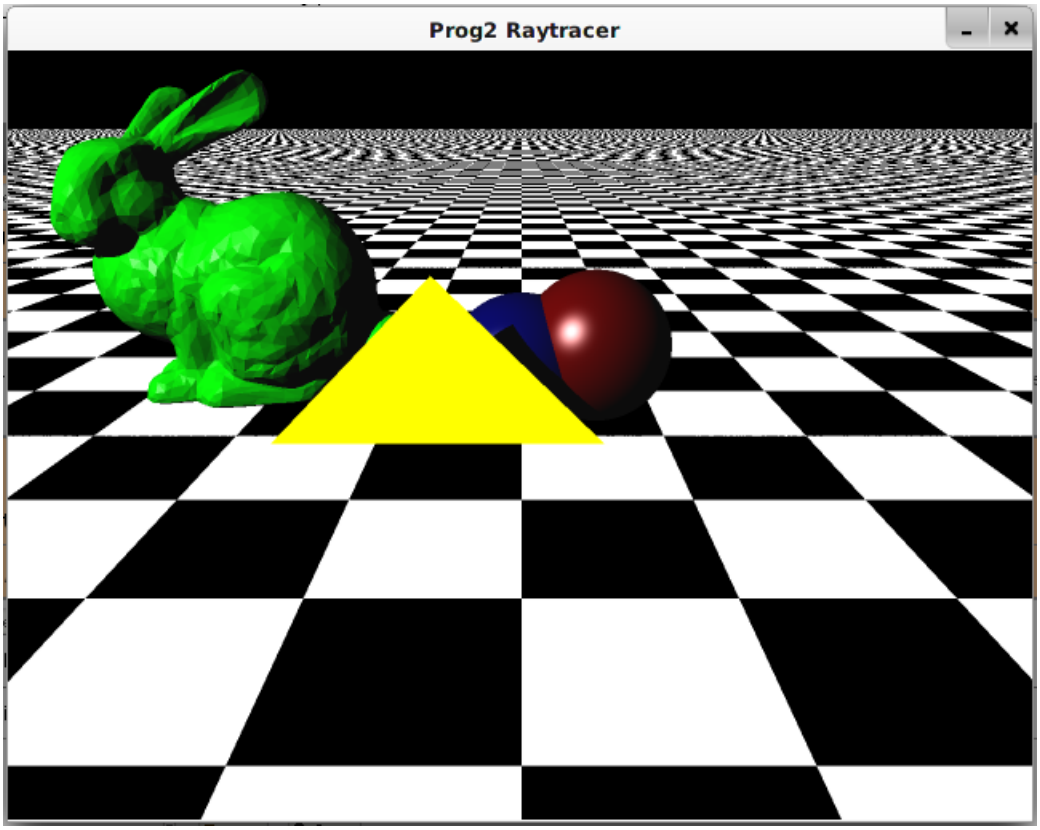

Figure 5: reference image