Rendering Pipeline
"Do It Yourself"

Computer Graphics, 2025-2026
2nd year, "Image and Multimedia" and "Software Engineering" tracks
Sylvie Chambon, Simone Gasparini et Géraldine Morin

Objectives

The goal of these four laboratory sessions is to **implement a rendering engine in `Java`**. More precisely, given a 3D object and a camera pose, you will render the image seen by the camera, as illustrated in Figure 1.

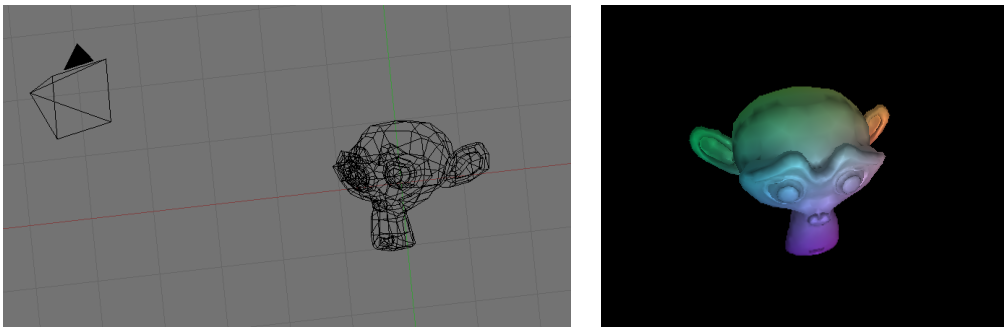


Figure 1: The objective of this assignment is to implement the rendering: (*Left*) input data consisting of a camera pose and a 3D object and the parameter file, (*Right*) rendered image generated from the camera viewpoint.

Evaluation

You are given four lab sessions to complete this assignment in pairs. **Your group/pair must be registered on Moodle during the first session** (here). The evaluation will take place both **individually** with a Moodle Quiz (March 13th, 16:15) and **as a group** with a code demonstration and an oral examination **during a dedicated session indicated in your timetable**. The exact time slot may vary between groups; please check your schedule carefully.

1 Description of the provided code and data

1.1 Camera and scene

- The **camera (position and orientation) and lighting parameters** are defined in a `*.scene` file. Empty lines and comments starting with # are ignored. This file contains:
  - the name of the file containing the object: `object_filename.OFF`;
  - the camera position: three coordinates $x_{cam}, y_{cam}, z_{cam}$;
  - the *LookAt*, the 3D point the camera is looking at: $x_{look}, y_{look}, z_{look}$;
  - the *up vector*, a vector indicating which direction is considered "up" for the camera, used together with the view direction to define the camera's coordinate frame (note that it does not necessarily correspond to the camera Y axis, but it allows for computing it);
  - the focal length f in pixels: in the camera coordinate system, the image plane is located at $Z = f$;
  - the image height h and width w in pixels (which are assumed to be square);
  - lighting parameters (described later in the assignment).

  Figure 2 shows all parameters contained in this file.

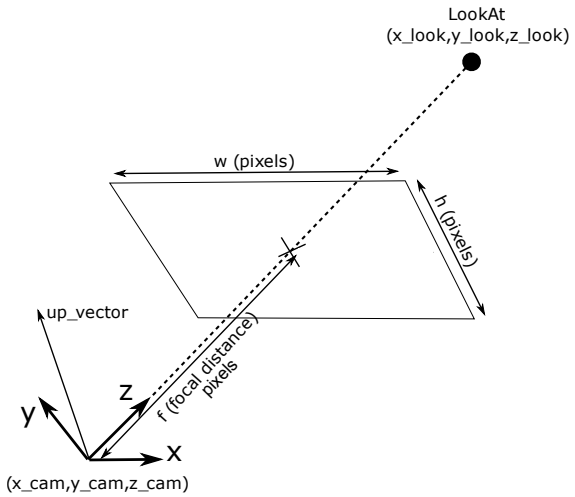


Figure 2: Parameters defined in the `*.scene` configuration file.

- **3D objects** are defined in a text file with the `*.OFF` format. A `*.OFF` file defines a **3D mesh** whose vertices each have a color and possibly texture coordinates. Let N_S, N_A, and N_F denote the number of vertices, edges, and faces, respectively. The file's structure is as follows:

Line number	File content
1	`OFF`
2	$N_S \ N_F \ N_A$
3	$x_0 \ y_0 \ z_0 \ R_0 \ G_0 \ B_0 \ x_{t_0} \ y_{t_0}$
...	...
$(3+N_S)$	$x_{N_S-1} \ y_{N_S-1} \ z_{N_S-1} \ R_{N_S-1} \ G_{N_S-1} \ B_{N_S-1} \ x_{t_{N_S-1}} \ y_{t_{N_S-1}}$
$(3+N_S+1)$	`3` $i \ j \ k$
...	...

For simplicity, we assume that each face is a triangle; therefore, the leading "3" indicates the number of vertices per face. Indices i, j, and k refer to vertex indices, defined by their order in the vertex list, which typically starts at 0.

Note: you are not required to implement the parsing of the `*.OFF` file. This is provided for a better understanding of the input data.

1.2 The program and its classes

The code provides the main framework within which you implement the rendering engine. The program is UI-based and enables you to test the code with various inputs and the different functionalities you will implement, as you can see in Figure 3

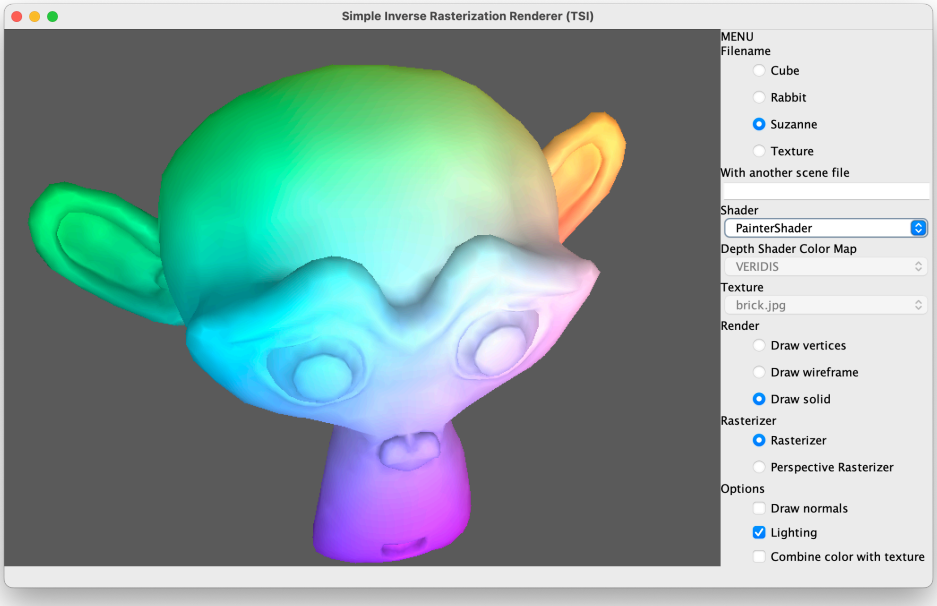


Figure 3: The program lets the user select the scene file and visualizes the 3D model with the different types of rendering that can be selected in the user interface.

The code follows the Model-View-Controller pattern (MVC), and it is organized accordingly. Figure 10 (the last page of this document) presents the overall software architecture. For this project, you will work mostly on the Model Layer.

Model Layer

The model layer contains the core rendering logic and data structures:

`renderer.algebra` contains mathematical classes and algebraic helpers to handle matrices and vectors {`Matrix, Vector, MathUtils`}`.java`

`renderer.core.mesh` contains classes for the 3D geometry representation:

- `Mesh.java`: Triangle mesh data structure with vertices, faces, normals. It loads a 3D mesh from a `.OFF` file. **Vertex coordinates, color components, and face vertex indices are stored sequentially**.

  For example, the vertex array is one-dimensional, of length $3 \times N_S$, and organized as $[x_0, y_0, z_0, x_1, y_1, z_1, \ldots, x_{N_S-1}, y_{N_S-1}, z_{N_S-1}]$.

- `Scene.java`: Scene description including camera, lights, and mesh references that can be loaded from a `.scene` file.

- `Texture.java`: Texture loading and sampling

`renderer.core.camera` Contains the class for the view transformations: `Transformation.java`: World-to-camera, projection, and calibration matrices

`renderer.core.light` Contains the classes that model the different types of lighting:

- `Light.java`: Abstract base class for the lights with its implementation for ambient and point light ({`AmbientLight, PointLight`}`.java`)

- `Lighting.java`: Manages scene lighting environment

`renderer.core.shader` Contains the classes for the Fragment processing (Shaders):

- `Fragment.java`: represents a pixel (a vertex on a 2D grid) together with its attributes (color, depth, normals, texture coordinates, *etc.*)

- `Shader.java`: Abstract shader base class with its implementations {`SimpleShader, PainterShader, TextureShader`}`.java`

- `DepthBuffer.java`: Z-buffer for depth testing

`renderer.core.rasterizer.*` Contains the classes for discretizing 2D geometric primitives (edges and faces) onto a pixel grid, and generating Fragments.

Controller Layer (`renderer.controller.*`)

These classes coordinate between model and view. The most interesting class here is `Renderer.java`. It represents the main rendering pipeline coordinator – loading scenes, projecting vertices, and rasterizing faces. Most of the code that you produce will be called by this class. **You do not have to edit this part of the code**.

View Layer (`renderer.gui.*`)

This package contains the user interface components. **You do not have to edit this part of the code**.

2 Getting started

Testing and getting familiar with the code

1. Download and unzip the archive from `Moodle`. The directory contains:
   - a `src` subdirectory containing all required Java classes (some of which must be completed), organized as described in Section 1.2
   - a `test` subdirectory containing test programs for the different classes;
   - a `data` subdirectory containing the scene and 3D model files.
2. To compile, use the command `make` from the root directory.
3. The code includes unit and functional tests. To run the tests, use `make tests`. All tests should compile and run without runtime errors.
4. To run program, use `make run`. At this stage, the renderer only displays a black image.
5. You can create the Javadoc for the code by running `make doc`.

   The documentation is created in the `doc` folder and can be viewed in a browser.

3 Image formation: from 3D to 2D

In this part, we will implement the transformations required to map a 3D vertex (`Vector`) to a point on the screen (`Fragment`).

Transforming a mesh vertex into an image point

The goal is to compute the pixel coordinates of the projection of each 3D vertex of the object onto the image plane. To achieve this, we incrementally implement the following functions:

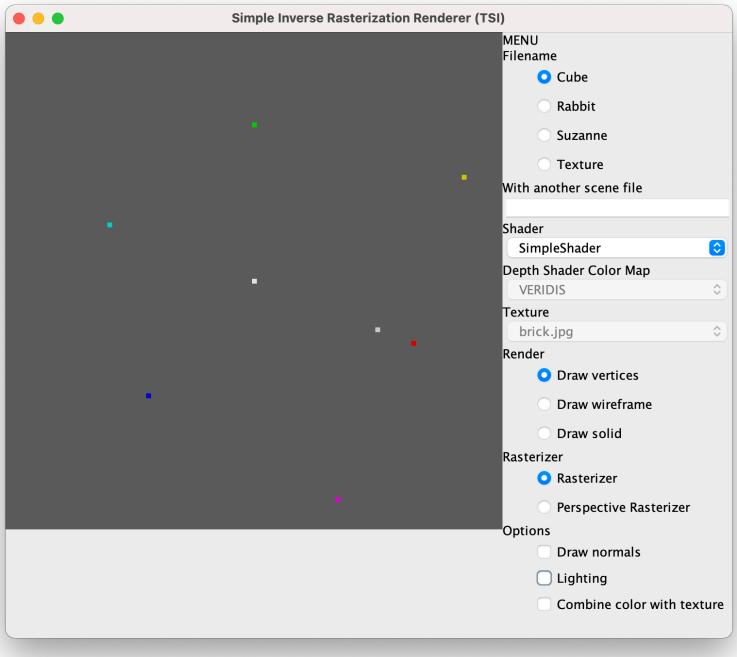
1. the function `setLookAt` (in `Transformation.java`), which defines the matrix `Transformation.worldToCamera`, as presented in the lectures;
2. the projection matrix in `setProjection`, stored in `Transformation.projection`;
3. the calibration matrix — a 2D/2D change of coordinate system to map to image coordinates — in `setCalibration`, stored in `Transformation.calibration`, also covered in the lectures;
4. the function `projectPoints`, which uses the three previous matrices to project a 3D vertex onto the image plane.

Verify that the depth of the projected pixel is stored in its attributes, as it will be required for depth buffering. This implies that **you must retain the z coordinate ($z \neq 1$)** when computing homogeneous coordinates.

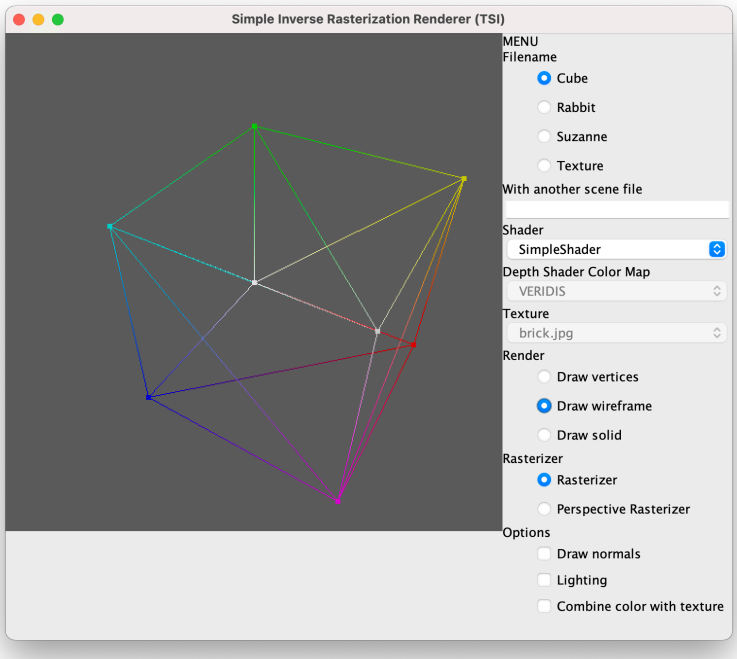
If everything is implemented correctly, you should now be able to visualize the projected vertices on the image, as shown in Figure 4a. You can also toggle the radio button between **Draw vertices** and **Draw wireframe**. The latter displays the edges of the mesh model (*i.e.* the wireframe) by interpolating the color between each pair of vertices connected by an edge (*cf.* Figure 4b).

Take a moment to read and understand the code associated with these two visualization modes, as it will be the base for the next step. In particular, the method `Rasterizer.RasterizeVertex()` draws a vertex with the color set in its attributes as a single pixel, slightly enlarged for better visibility.

The method `Rasterizer.RasterizeEdge()` draws each edge as a line segment using the **Bresenham algorithm**. The attributes of each pixel on the line are linearly interpolated between the segment endpoints using the `interpolate2` function.



(a)



(b)

Figure 4: Once the transformations are correctly implemented, the renderer displays the colored vertices, **Draw vertices** (a), or the wireframe rendering of the mesh, **Draw wireframe** (b).

4 Triangular face rasterization

Face rasterization

Now it is time to fill the faces of the model as shown in Figure 5b. We are going to implement a first solid rendering method of a face by linearly interpolating, in 2D, the attributes of the vertex pixels (*Fragments*) of the face (we are assuming it is a triangle). The goal is to fill a triangle defined using its three vertices, represented as Fragments, by completing the method `Rasterizer.rasterizeFace()`. Filling consists of determining all pixels inside the triangle and computing their color via interpolation: **all fragment attributes must be interpolated linearly**. Moreover, due to numerical approximations, be sure that the interpolated color values are always in the range $[0, 1]$. For that, you can use the clamp function `MathUtils.clamp()`
First, determine which pixels p lie inside the triangle. This requires:

- using the vertex coordinates to determine the ranges of variation in x and y;
- retaining the pixels p for which **all barycentric coordinates are positive**.

The barycentric coordinates (α, β, γ) of a point $p = (x, y)$ in the basis (x_i, y_i), $i = 1, 2, 3$, can be computed using the matrix $\mathbf{C}$ given by the method `Rasterizer.makeBarycentricCoordsMatrix()`. (*cf.* the beginning of the `rasterizeFace` method). You need to compute the barycentric coordinates (α, β, γ) of $p = (x, y)$ and interpolate the attributes (weighted sum using barycentric coordinates):

$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \mathbf{C} \begin{pmatrix} 1 \\ x \\ y \end{pmatrix}, \tag{1}$$

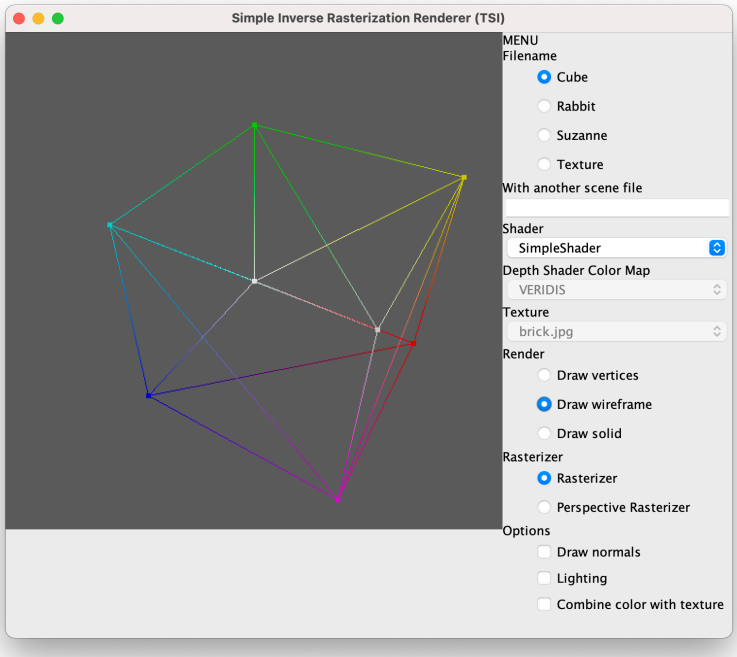
with

$$\mathbf{C} = \frac{1}{A} \begin{pmatrix} x_2y_3 - x_3y_2 & y_2 - y_3 & x_3 - x_2 \\ x_3y_1 - x_1y_3 & y_3 - y_1 & x_1 - x_3 \\ x_1y_2 - x_2y_1 & y_1 - y_2 & x_2 - x_1 \end{pmatrix}, \tag{2}$$

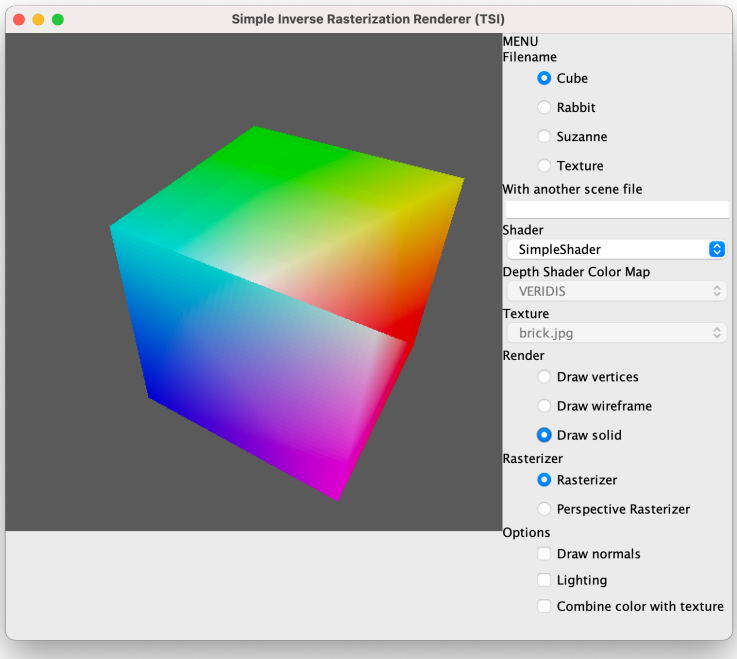
where A is twice the area of the triangle:

$$A = x_2y_3 - x_3y_2 + x_3y_1 - x_1y_3 + x_1y_2 - x_2y_1.$$

After interpolating the fragment, display it directly by calling the shader: `shader.shade(your_fragment)`.



(a)



(b)

Figure 5: Rendering results obtained with the cube with (a) **Wireframe rendering** and (b) **Solid rendering** without depth buffering.

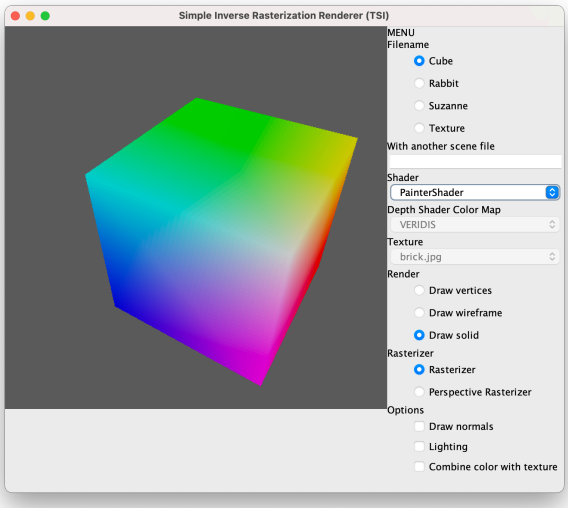


Figure 6: Result with the cube model after adding the depth buffering with the z-buffer.

5 Depth buffering

To correctly handle the depth, *i.e.* to display only visible surfaces and hide those occluded by closer faces, the depth information must be preserved and taken into account. When displaying a pixel, a check must be added to verify whether it has already been drawn. If so, the depths of the current and previous corresponding 3D points must be compared, and the one closest to the camera must be preserved.

Depth buffer

Depth buffering is achieved by storing, in a depth buffer, the depth of each pixel, corresponding to the depth of the 3D point currently displayed at that pixel.

1. To do so, in the `DepthBuffer` class, complete the methods `testFragment` and `writeFragment`.
2. The shader `PainterShader` uses the `DepthBuffer` to store depth information for each pixel. Check its method `PainterShader.shade()` to see how it updates the pixel if the corresponding Fragment satisfies the test.
3. Once done, run the program and select the **PainterShader** in the combo box Shader of the UI.

You should now obtain a result that correctly accounts for depth, as shown in Figure 6.

6 Lighting – Gouraud shading

As seen in the lectures, two lighting algorithms can be implemented: Gouraud shading or Phong shading. Here, we first implement Gouraud shading, which is simpler and computationally cheaper.

The parameters required for lighting are specified in the **.scene* file:

- ambient light intensity: I_a;
- a point light defined by its homogeneous coordinates $x\ y\ z\ w$ (allowing the light to be placed at infinity. *i.e.* $w = 0$) and its intensity I_d;
- the material coefficients of the object: K_a, K_d, K_s, s.

Computing light intensity at each vertex

1. To account for lighting in the Gouraud algorithm, surface normals must be computed to derive intensity. Implement the normal computation for each vertex in the method `computeNormals` of the `Mesh` class. The normal at a vertex is obtained as the average of the normals of the faces incident to that vertex (note that the resulting vector must be normalized).
2. Verify that the computation is correct by enabling the checkbox **Draw normals** in UI. The renderer will draw the normal to each vertex as a red segment (*cf*. Figure 7).
3. The light intensity is computed in the class `Lighting`, using the method `applyLights()`. The class contains a list of the lights declared in the scene. The method is called for each vertex, and it sums the contribution (`getContribution()`) of each light, according to the formula introduced in the lectures:

   $$I = I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}},$$

   where

   (a) $I_{\text{ambient}} = I_a \cdot K_a$;
   (b) $I_{\text{diffuse}} = I_d \cdot K_d \cdot \cos(n, l)$, where n is the surface normal and l the light direction;
   (c) $I_{\text{specular}} = I_d \cdot K_s \cdot \cos^s(h, n)$, where h is the bisector of l and v, the view direction.
4. Complete the implementation of the `getContribution()` methods for the classes modeling the different types of lights (*i.e.* Ambient and Point).

The method `applyLights()` is called in `Renderer` via the method `projectVertices`, which itself is called in `renderSolid`. For each point inside a face, the intensity is obtained as an affine combination of the vertex intensities of the face (using the fragment interpolation implemented in Section 4).
Finally, enable the option **Lighting** in the UI as shown in Figure 8 to see the effect of the lighting on the monkey model.

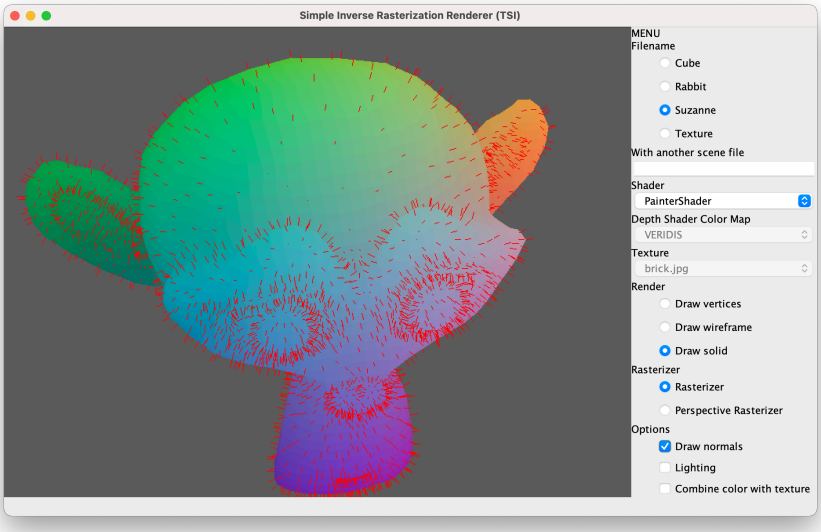
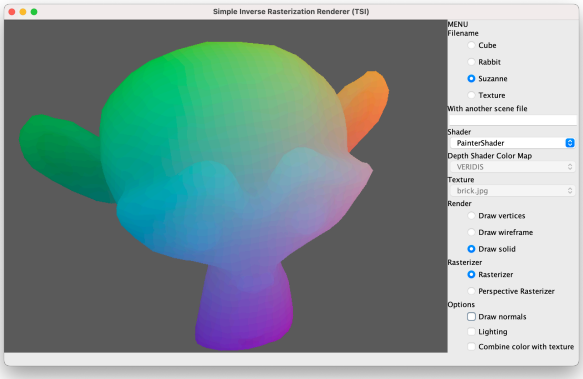
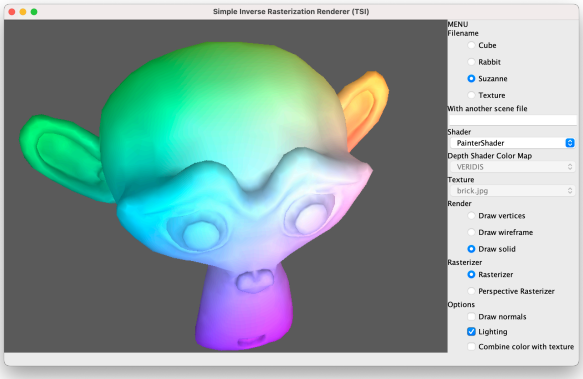


Figure 7: The option **Draw normals** draws the normal of each vertex with a red segment.



(a) **Lighting** disabled.



(b) **Lighting** enabled.

Figure 8: Rendering result with the monkey model: before lighting (a) and after lighting (b).

7 Texture mapping

In this final part, we will apply textures to a 3D object. The texture coordinates are defined for each vertex in the OFF file (*cf*. Section 1). You should therefore use the example `example_textured.scene` (**Texture** filename in the UI).

Texture coordinates describe how a 2D texture image is mapped onto the surface of a 3D object. Each fragment stores texture coordinates (u, v) that indicate which location of the texture image should be sampled to determine its color. In general, texture coordinates (u, v) are defined in the range $[0, 1]$, where u and v represent normalized positions w.r.t. the image width and image height, respectively.

You can open `textured_facet.off` in the editor to check the values of u and v for this model (the last two values of each line). You should notice that, in this case, the texture coordinates are not restricted to the range $[0, 1]$. This is because a single, smaller texture image can be efficiently reused and repeated multiple times over the same face, which is a common technique in computer graphics known as texture tiling. This behavior can be observed for the brick pattern shown in Figure 9a.

Computing the pixel color from the texture

The attributes corresponding to texture coordinates are stored in the fragment attributes. They can be accessed using the method `getAttribute(int, int)` in `Fragment.java` (see the method documentation).

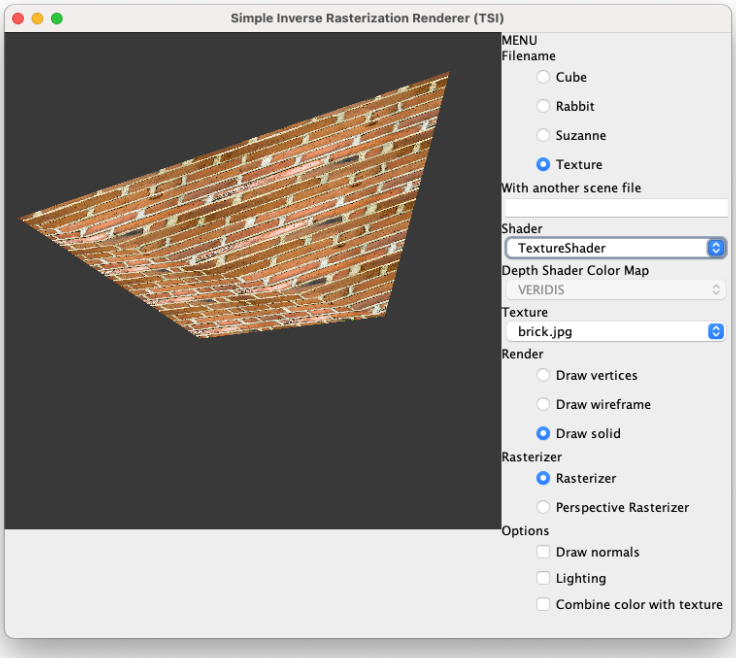
1. Complete the method `Texture.sample(double u, double v)` that samples the texture image at the texture coordinates (u, v). As explained above, (u, v) are normalized with respect to each image dimension and may be greater than 1 when the texture is repeated over a face. This must be taken into account when determining the location in the texture image from which the color is sampled.

2. Complete the method `TextureShader.shade()`

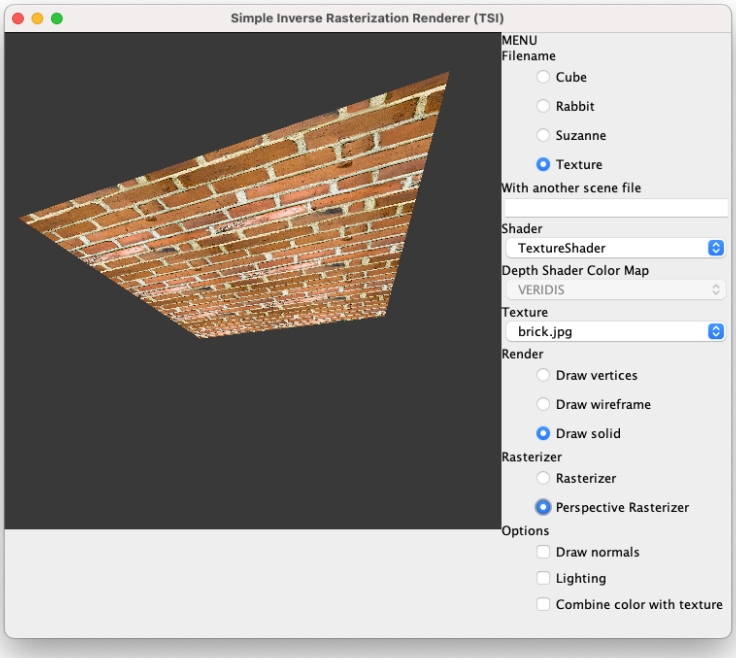
Note that `TextureShader` does not work unless the `depthBuffer` is implemented. Once everything is implemented, select the TextureShader in the UI to display the texture as shown in Figure 9a.

The resulting rendering does not look correct because of the incorrect perspective distortion. To address this issue, another class `PerspectiveCorrectRasterizer.java` is provided, which corrects the perspective. Check the code of the class to understand how it works. Then you can change the rasterizer in the radio button of the UI by selecting **PerspectiveRasterizer**.

Figure 9 illustrates the effect of this correction on the textured wall example.



(a) Result with the `Rasterizer`



(b) Result with the `PerspectiveCorrectRasterizer`

Figure 9: Textured rendering before perspective correction (a) and after correction (b).

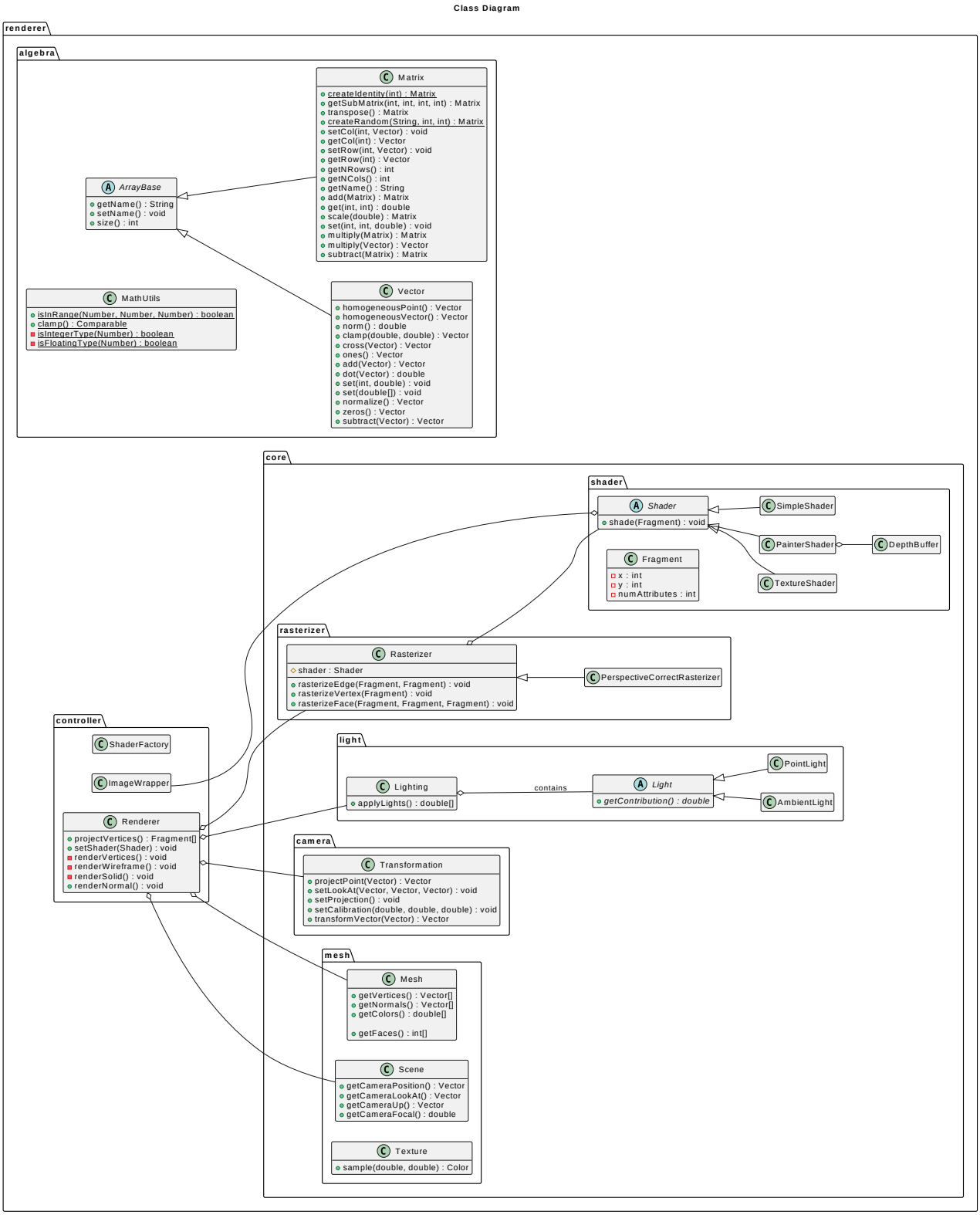


Figure 10: UML class diagram of the renderer.