Lab I: Exercise Pipes & Filters

The task in this lab exercise is to implement a simple variant of the computer graphics pipeline both as *push* **and** *pull* pipeline using the pipes and filters architecture. More specifically you have to implement perspective 3D **software** rendering of the famous Utah Teapot rotating around its Y-Axis in four different styles and colors, see Figure 1.1.

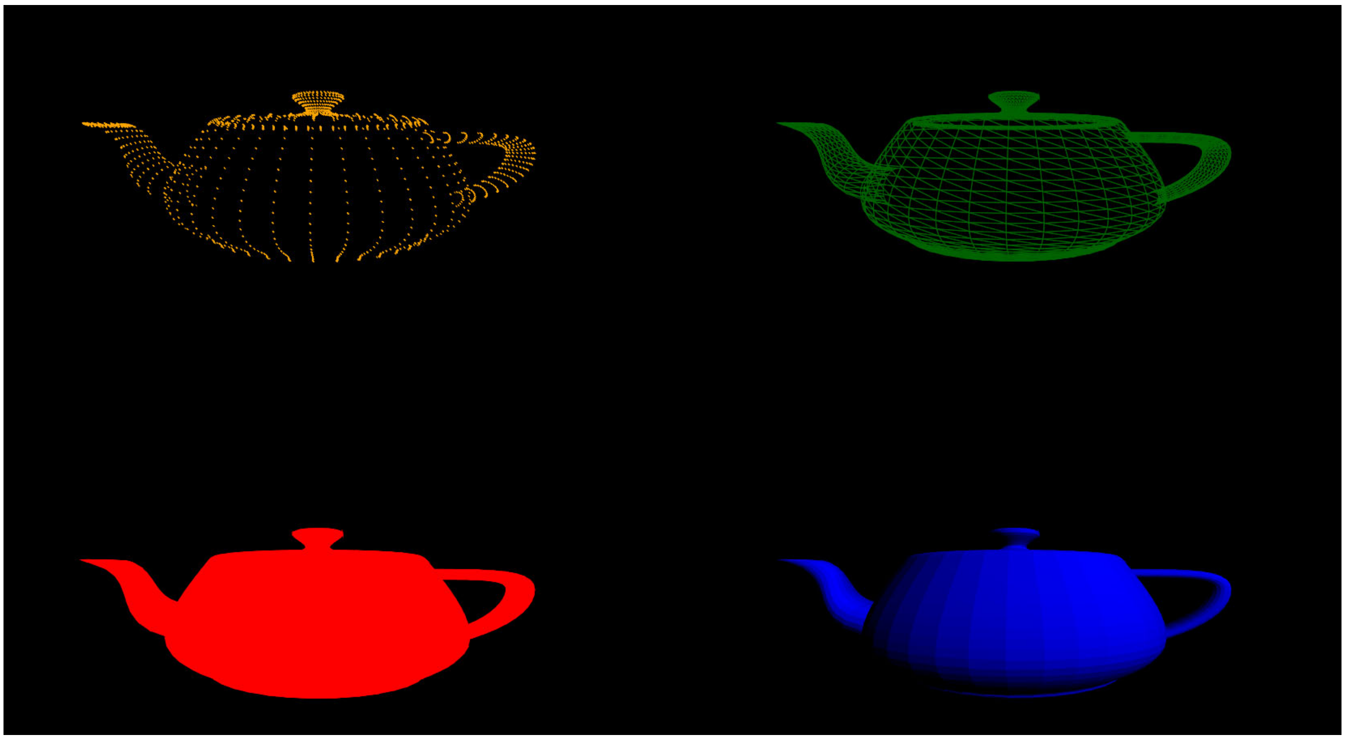

Figure 1.1: Screenshot of the Lab 3 reference implementation

The four different styles are as follows:

1. Point rendering in orange colour in top left corner.
2. Wireframe rendering in dark green colour in top right corner.
3. Filled rendering in dark red colour in the bottom left corner.
4. Filled, shaded rendering in blue colour in the bottom right corner.

In more technical terms, you have to implement the following:

1. Model-View transformation for rotation around the Y-Axis.
2. Backface culling in view space of triangles facing away from the camera. This has to be done as early as possible in the pipline to reduce unnecessary computing.

3. Depth sorting in view space for improved visibility in one of the two pipelines (you need to figure out in which).
4. Individual coloring of the teapots.
5. Simple lighting in view space with the flat shading model.
6. Perspective projection and screen space transformation.
7. Different rendering styles of polygons as points, wireframes and filled.

All these features are to be implemented in software without the use of a GPU. However, for the sake of simplicity (and performance) this pipeline is focusing mainly on the Geometry Processing Stage, does not perform clipping, does not perform per-pixel computations such as per-pixel shading and leaves rasterization to JavaFX, which also deals implicitly with clipping. More specifically, you should only operate on triangles and additional properties (for example color) which are transformed in multiple steps and then passed to a JavaFX canvas for rasterization. Therefore, we arrive at a simplified graphis rendering pipeline, and in this exercise we recommend that it follows the transformation steps as shown in Figure 1.2.

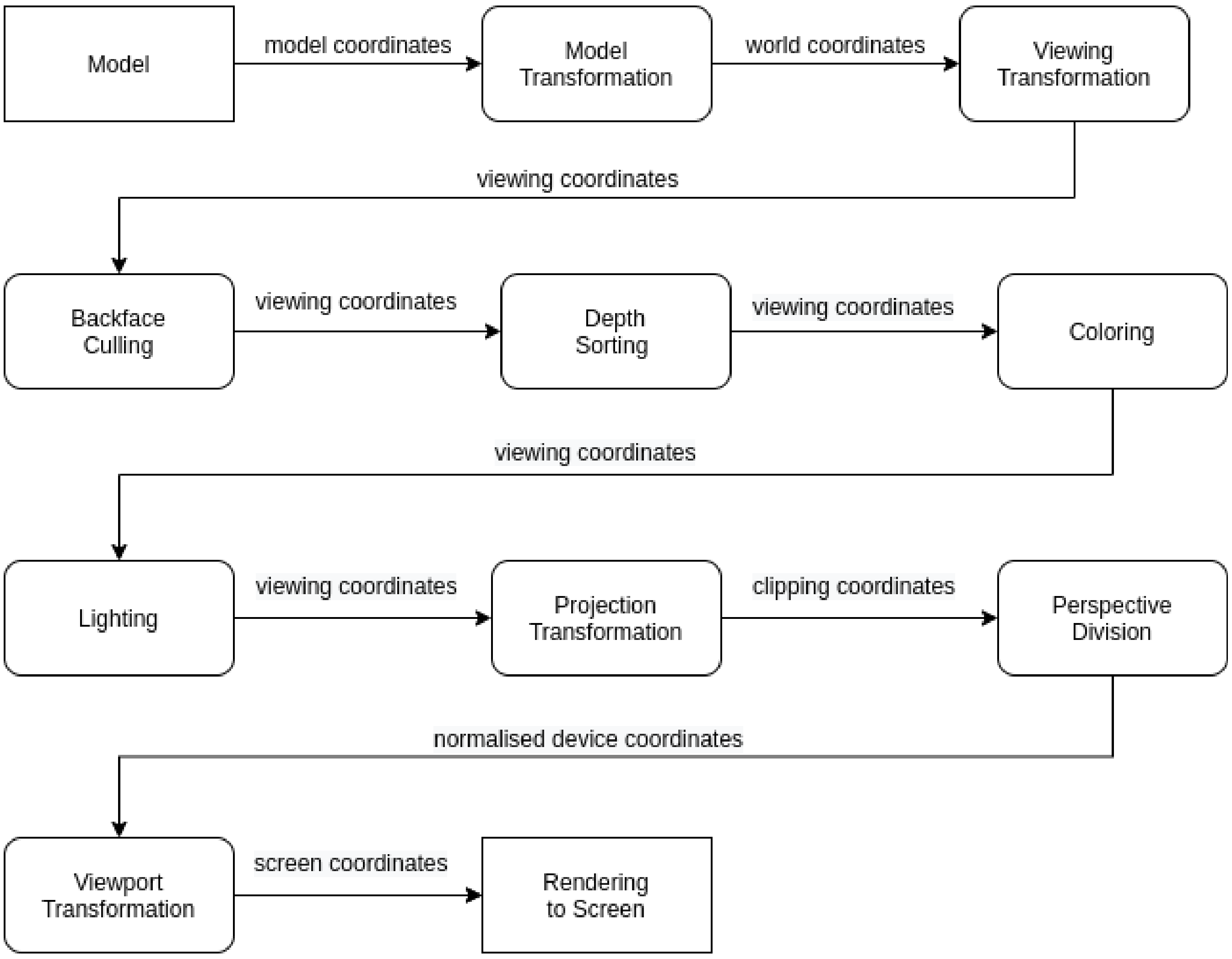

Figure 1.2: The transformation pipeline of this exercise.

1.1 Implementation

You don’t have to start from scratch. Please download the Code Skeleton in Ilias which provides the following for you:

- Steps to implement as TODOs.
- Setting up the JavaFX Scene with four Canvas elements.
- A simple animation framework in JavaFX which is used for driving the rendering and animation of the teapot.
- Model loading.
- Scene configuration with modeling, viewing, projective and viewport transformations already set up.
- Included the *jglm* library as sources for OpenGL conformant matrix and vectors and transformation operations (multiplication, subtraction, normalisation, cross product, dot product). You will normally only going to use the classes `Mat4` , `Vec3` , `Vec2` and `Matrices` . Therefore you do not to have implement your own Matrix and Vector classes and the respective transformations.

It is recommended to implement the rendering pipeline first with either push or pull but both not at the same time. This allows you to develop an understanding of both the rendering pipeline and Pipes and Filters. The implementation of the other pipeline should then follow similar principles and should be done a lot faster. There exist the `PullPipelineFactory` and `PushPipelineFactory` classes, which act as the starting points for you. In there you will find specific TODOs which help you in implementing the pipeline. **You are not allowed to change the code of the skeleton with the exception of the two classes above**.

Make sure to define general interfaces for pipes and filters for each of the two pipeline types. Maybe you will also find implementing an abstract helper class for the filter or pipe useful. Use generics for compile-time type safety, which prevents that only compatible pipes and filters can be connected. You might need the `?` wildcard for Java generics which indicates an unknown type. Use it when a generic class does not or should not depend on the specific generic type, for example:

```java
class ToyGenericExample<T> {
    // This specific generic type T needs to be part of the generic class definition.
    private List<T> listOfTs;

    // This is an unknown, wildcard generic type, therefore can be ANY generic type
    // and does not have to be part of the generic class definition.
    private List<?> listOfUnknowns;
}
```

1.2 Model-View Transformation

The teapot has to rotate around its Y-Axis. The transformation matrix, which places the model into the world is already provided for you in the `PipelineData.getModelTranslation` . Based on this you have to create a new rotation matrix in each frame to create a new modeling matrix based on the translation matrix. To create the correct rotation matrix, use `Matrices.rotate` with the corresponding rotation vector `PipelineData.getModelRotAxis` and rotation angle (in radians). The resulting rotation matrix is then used to create a new model-view transformation, where the viewing transformation can be accessed from `PipelineData.getViewTransform` . You need to think about how to deal with a continuous rotation, which is independent from the framerate. Also please remember that matrix multiplication is **not** commutative, therefore you have to mulitply the matrices in the correct order!

1.3 Depth Sorting

It is explicitly **not** the task to implement a depth buffer, which would require to implement your own rasterizer, which goes far beyond the scope of this exercise. However, it is mandatory to implement depth sorting for a (mostly) correct visibility. Depth sorting simply sorts the faces in view space according to their z (depth) value back to front, descending with the z value, that is the face with the highest z value is the first. This has the effect that we render the faces back-to-front, which results in faces closer to the camera to occlude the faces farther away (aka *Painters Algorithm*). You need a single z value of each face, for sorting purposes, therefore compute the average of all z values of a face which gives a good result compared to its computational effort (other options are min/max or simply picking a fixed vertex).

This approach is only reasonable possible in one of the two pipelines and you need to figure out whether it is best implemented in the *push* or *pull* pipeline. Therefore, either one of the rendering pipeline lacks correct visibility testing, which leads to artefacts, as can be seen in Figure 1.3.

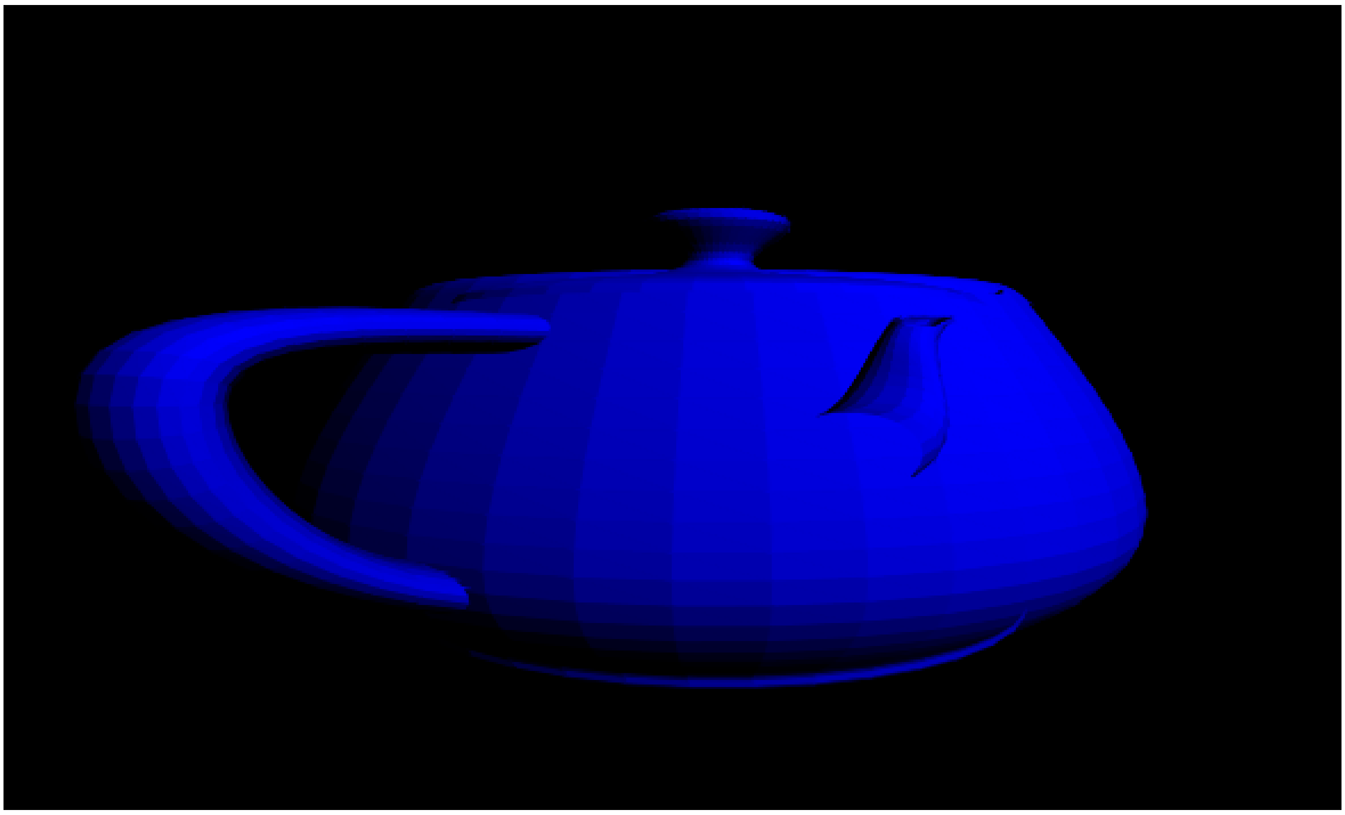
Figure 1.3: Visibility artefacts without depth sorting.

In this figure artefacts are visible where polygons appear to be in front of other polygons which should be invisible as they have a higher z (depth) value. Note that due to overlapping and imprecision you will also see a few artefacts when using depth sorting, see Figure 1.4.

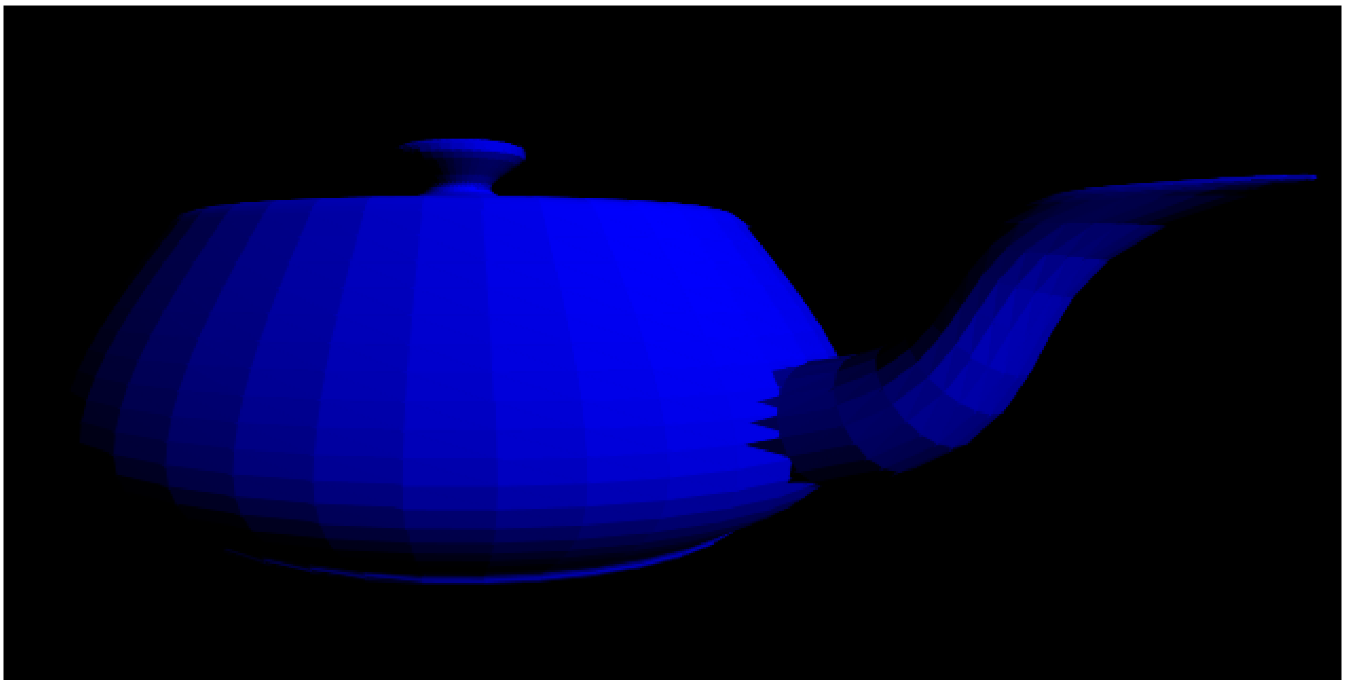
Figure 1.4: Visibility artefacts with depth sorting.

1.4 Backface Culling

You have to implement backface culling, which is trivial when the Face is in VIEW space. In general you simply need to compute the dot product between the vertex and its normal and if it is larger than 0 the face has to be culled, that is not processed any further in the pipeline, see Figure 1.5.

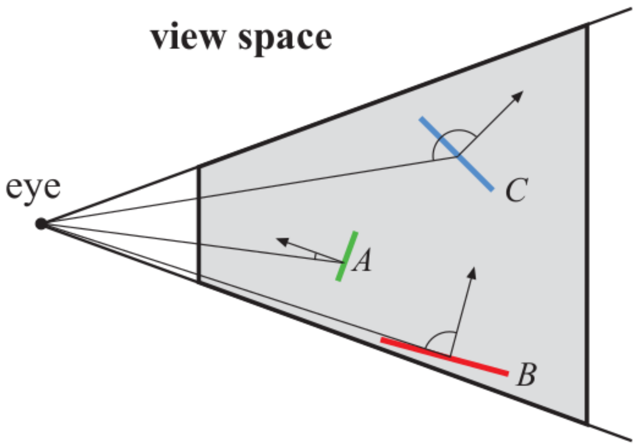

Figure 1.5: Backface culling (Figure taken from [2]).

Depending on the winding order, the normals point inwards or outwards, requiring a different handling either larger than 0 or less than 0. However in the teapot model provided for this exercise the normals are provided for you and you can assume the following:

$$V_1 \cdot N_1 > 0 \Rightarrow cullface$$

Backface culling leads to considerable performance improvements, as it removes roughly 50% of all triangles which means they do not have to be processed by transformations further down the pipeline. In the reference implementation the performance improves about 400% (each canvas rendering pipeline performance roughly doubles as it only needs to render roughly half of the polygons). See Figure 1.6 for a screenshot where backface culling is turned off.

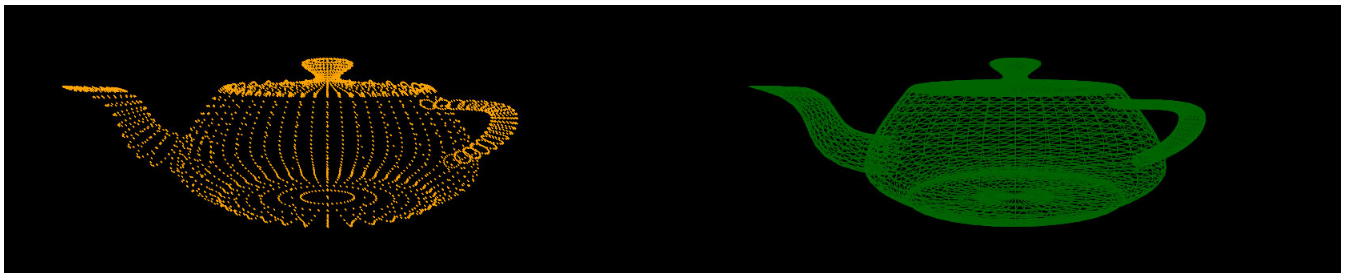
Figure 1.6: Disabled backface culling.

1.5 Individual Colouring

All four teapots have to have different colors, provided through `PipelineData.getModelColor` . You need to think about how to pass color information along with the pipeline data.

1.6 Flat Shading

In the lower right corner of Figure 1.1 you see the filled, flat shaded rending of the teapot. You have to implement the simple flat shading model, of diffuse lighting which operates in *view space*. Diffuse lighting gives the object more brightness the closer its fragments are aligned to the light rays from a light source. To give you a better understanding of diffuse lighting, see Figure 1.7.

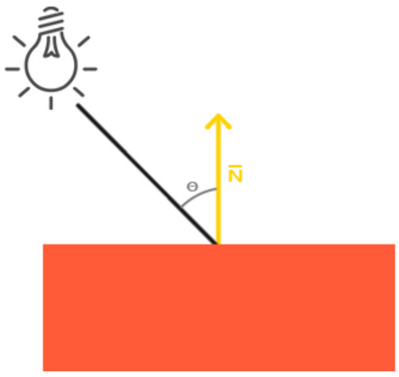

Figure 1.7: Diffuse lighting (Figure taken from https://learnopengl.com/Lighting/Basic-

Lighting).

To the left is a light source with a light ray targeted at a single fragment of our object. You need to measure at what angle the light ray touches the vertex. If the light ray is perpendicular to the object's surface the light has the greatest impact. To measure the angle between the light ray and the vertex use the normal vector (which is provided for each vertex, see the `Face` class), that is the vector perpendicular to the fragment's surface (here depicted as a yellow arrow). The angle between the two vectors can then easily be calculated with the dot product.

Therefore, you simply need to calculate the normalised normal vector between the face and the light position, which you can obtain from `PipelineData.getLightPos` . With this normal you compute the dot product between the face normal. The dot product of two normalised vectors is always between -1 and 1, corresponding to the cosine of the angle between them, see Figure 1.8.

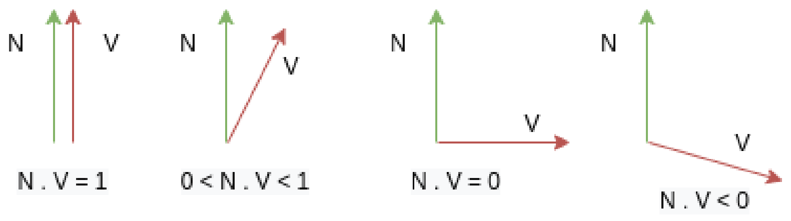

Figure 1.8: The dot product between two normalised vectors.

More specifically the dot product satisfies the following:

$$\overline{v} \cdot \overline{k} = ||\overline{v}|| \cdot ||\overline{k}|| \cdot cos\,\theta$$

If both v and k have unit length this results in the very convenient identity:

$$\overline{v} \cdot \overline{k} = 1 \cdot 1 \cdot cos\,\theta = cos\,\theta$$

Using the dot product you have to attenuate the color of the face. Therefore the more direct the income of the light the lighter the face will be. If the dot product is below or equal zero it means that the face is not facing the light and is therefore black (assuming no other indirect light).

Note that only this single teapot is rendered with flat shading, therefore you need to check `PipelineData.isPerformLighting` when constructing the pipeline and omit it in case not. It is possible to apply flat shading also to the other rendering modes, however the effect will not be as pronounced, see Figure 1.9.

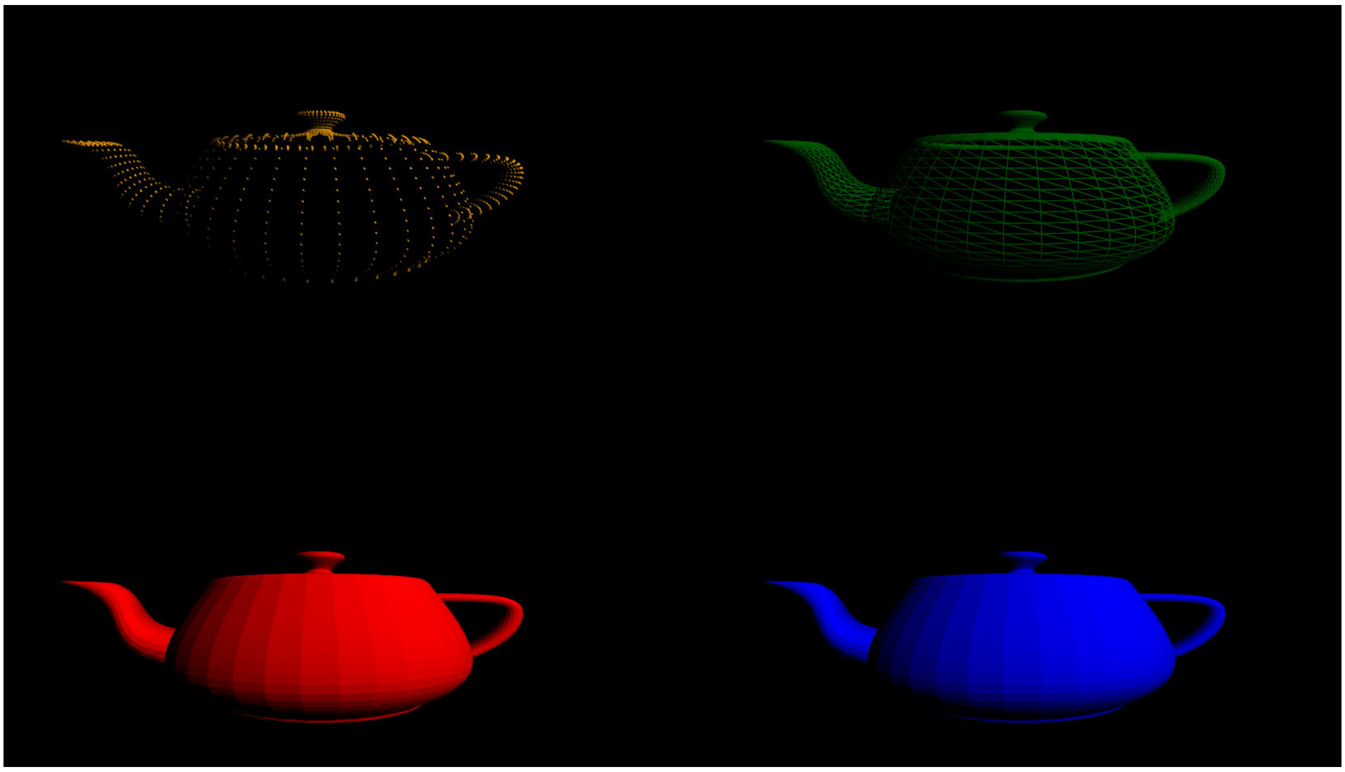
Figure 1.9: Flat shading in all rendering styles.

1.7 Perspective Projection

As it is a rendering of a 3D model, you also need to perform a perspective projection, where in this case we use the projective transformation. The transformation is already set up for you and can be accessed from `PipelineData.getProjTransform` . You only need to transform the faces in view space using this transformation, with the face then ending up in clipping coordinates.

Note that we do **not** perform clipping in this exercise as it goes beyond its scope and we leave the actual rendering and clipping to JavaFX.

1.8 Screen Space Transformation

Having transformed the faces into clipping space, you now need to transform them into 2D screen space. This is simply done by performing the perspective division by the `w` component of each `Vec4` vertex and then transforming the resulting vertices using the viewport transformation into screen space. The viewport transformation is also already set up for you and can be accessed from `PipelineData.getViewportTransform` .

1.9 Rendering Styles

The resulting screen space coordinates (`x` and `y` component) of the `Vec4` can then be passed as `Vec2` , using `Vec4.toScreen` to the JavaFX `GraphicsContext` , available from `Pipeline.getGraphicsContext` . Using this you can:

- Render points with `GraphicsContext.strokeLine` and setting the color with `GraphicsContext.setStroke` .
- Render wireframes with `GraphicsContext.strokePolygon` and setting the color with `GraphicsContext.setStroke` .
- Render filled with `GraphicsContext.fillPolygon` and setting the color with `GraphicsContext.setFill` .

References

[2] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. 2019. *Real-time rendering*. Crc Press.