WIRTSCHAFTSUNIVERSITÄT WIEN
Vienna University of Economics and Business

Bachelor's Thesis

Titel of Bachelor's Thesis (english)	An Introduction to JavaFX 3D with BSF4ooRexx
Titel of Bachelor's Thesis (german)	Eine Einführung in JavaFX 3D mit BSF4ooRexx
Author (last name, first name):	Steger, René
Student ID number:	00950565
Degree program:	Bachelor of Science (WU), BSc (WU)
Examiner (degree, first name, last name):	ao.Univ.Prof. Mag.Dr.rer.soc.oec. Rony G. Flatscher

I hereby declare that:

1. I have written this Bachelor's thesis myself, independently and without the aid of unfair or unauthorized resources. Whenever content has been taken directly or indirectly from other sources, this has been indicated and the source referenced.
2. This Bachelor's Thesis has not been previously presented as an examination paper in this or any other form in Austria or abroad.
3. This Bachelor's Thesis is identical with the thesis assessed by the examiner.
4. (only applicable if the thesis was written by more than one author): this Bachelor's thesis was written together with

   The individual contributions of each writer as well as the co-written passages have been indicated.

14/07/2023
Date

Unterschrift

BACHELOR THESIS

An Introduction to JavaFX 3D with BSF4ooRexx

Author

René Steger

Supervisor

ao. Univ. Prof. Mag. Dr. Rony G. Flatscher

Vienna University of Economics and Business

Abstract

The aim of this thesis is to show how 3D can be used in JavaFX with the help of ooRexx and BSF4ooRexx. Nutshell examples are used to illustrate how 3D objects can be created and their properties defined. Furthermore, this work includes installation instructions for the required programmes and an explanation of them. ooRexx is an object-oriented programming language which, in combination with BSF4ooRexx, makes it possible to build a bridge to Java and thus use its libraries for programming.

Table of Contents

List of Figures

List of Listings

List of Abbreviations

AWT	Abstract Window Toolkit
BSF4ooRexx	Bean Scripting Framework for ooRexx
GUI	Graphical User Interface
IDE	Integrated Development Environment
JRE	Java Runtime Environment
JVM	Java Virtual Machine
ooRexx	Open Object Rexx
Rexx	Restructured Extended Executor
RexxLA	Rexx Language Association
SAA	System Application Architecture

1 Introduction

This bachelor thesis examines how a bridge can be built to Java by means of ooRexx in combination with BSF4ooRexx and thus 3D objects can be created in JavaFX.

The main goal, besides a theoretical introduction to the required software components, is to program nutshell examples, demonstrate the output and explain it. However, the purpose of the work is not to provide more complex code examples, but to demonstrate the basic functions of JavaFX, such as the creation of 3D shapes with the Java classes available for this purpose. Based on this, the various possibilities for defining and changing the properties of the generated objects will be presented.

The following chapter deals with the required software components. For this purpose, a short overview of the programming languages ooRexx and Java is given and it is explained in more detail how BSF4ooRexx builds a bridge from ooRexx to Java.

Chapter 3 deals with the installation of the required programmes. It shows step by step what to do if, for example, older versions are already installed on the operating system.

Chapter 4 shows the nutshell examples programmed in this bachelor thesis, which make up the main part of this paper. They are described using the written code and pictures of the output.

Finally, in the last chapter, the work is discussed once more and it is explained what could still be programmed in a further occupation with JavaFX.

2 Used Software

This chapter introduces the software components used in this thesis and provides information about them. On the one hand, these are the two programming languages Java and ooRexx and, on the other hand, BSF4ooRexx. Furthermore, the optional components in the form of IntelliJ and the corresponding ooRexx plugin are discussed. Instructions for installing this software can be found in the following chapter.

2.1 Rexx / ooRexx

Mike F. Cowlishaw developed a programming language in IBM's research facilities in 1979 that was easier for humans to understand and program than the dubious mainframe batch language Exec 2 used at IBM until then (Flatscher, 2013). This was the programming language Rexx, which in 1987 became the scripting and batch language for IBM for all its operating systems, after the SAA (System Application Architecture) had been introduced earlier (Flatscher, 2006). Subsequently, numerous open source and commercial Rexx interpreters, which did not originate from IBM, were developed, indicating the growing popularity of Rexx (Flatscher, 2013).

At the end of the 1980s, a project was started with the aim of expanding the Rexx programming language to include object-oriented functions (Flatscher, 2013). This project became Object Rexx, which was distributed for the first time in 1997. IBM's own versions of Object Rexx were created for IBM's AIX and Microsoft Windows (Flatscher, 2013).

In 2005, the Rexx Language Association (RexxLA) released Open Object Rexx, the open-source version of Object Rexx, after IBM gave them the source code in 2004 (Flatscher, 2013).

With ooRexx classic Rexx programs can be executed and written, as it is a classic Rexx interpreter (Flatscher, 2013). ooRexx is available for various operating systems in 32- and 64-bit versions. Programs created on one operating system can run on any other for which ooRexx is available (Flatscher, 2013). ooRexx also has an excellent documentation, as this has also been passed on to RexxLA by IBM in addition to the source code and can thus be kept up to date (Flatscher, 2013).

Some advantages of ooRexx are:

- English-like language: Unlike other programming languages, ooRexx uses common English words for instructions instead of abbreviations and symbols. This makes ooRexx easier to learn and use.
- Fewer rules: There are few formatting rules in ooRexx. Instructions can be written in upper or lower case and can span several lines. Likewise, several instructions can be on the same line.
- Interpreted, not compiled: Since ooRexx is an interpreted language, unlike other programming languages, programmes do not need to be compiled before execution.
- Clear error messages: When a programme encounters an error, ooRexx issues a message explaining the error in a meaningful way.

(Rexx Language Association, n.d.)

2.2 BSF4ooRexx

BSF4ooRexx, the Bean Scripting Framework for ooRexx, is an external Rexx function package that enables ooRexx to interact directly with Java (Flatscher, 2012). It uses the open-source class library BSF (Bean Scripting Framework) of Java, with which scripting languages can be implemented in Java (Flatscher, 2012).

By masking the entire class library and interactions with objects of Java, the impression was created that these are in fact ooRexx libraries and objects. This makes the interaction with Java easier to understand for ooRexx programmers (Flatscher, 2012).

Using BSF4ooRexx has several advantages for programmers, which are listed here:

- If the required functionality is already available in the Java Runtime Environment (JRE), no further external function packages are necessary.
- All Java classes of the JRE can be used which are executable on the operating systems supported by Java in their present form.
- Any Java class libraries can be used directly.
- Abstract methods can be used if they are implented in ooRexx, thus realising a callback mechanism from Java to ooRexx.
- Independent of the installed operating system, all information systems can be controlled via ooRexx, provided they have Java programming interfaces.

- Java applications are able to execute Rexx scripts under their own control.

(Flatscher, 2012)

2.3 Java

In 1995, Sun Microsystems published the Java programming language. It follows the paradigm of object-oriented programming and is class-based (Wikipedia, 2023-b). With Java, once compiled Java code has been written, it should be possible to run it on any system that has Java support without recompilation (Wikipedia, 2023-b). Normally, Java applications are compiled into bytecode. The syntax is in the style of C and C++, but there are fewer low-level functions available (Wikipedia, 2023-b). In 2007, Sun Microsystems' Java Virtual Machine was made available as open-source software. With the takeover of Sun Microsystems by Oracle in 2010, they also took over the development of Java (Wikipedia, 2023-b).

There were five main objectives in the development of the Java language. It should be:

- simple, object-oriented and familiar
- robust and secure
- architecture-neutral and portable
- efficient
- interpretable, dynamic and threaded.

(Wikipedia, 2023-b)

2.4 JavaFX

JavaFX is a framework for creating Java applications that was published in 2008. It is the successor to AWT (Abstract Window Toolkit) and Swing (Wikipedia, 2022).

AWT already existed in the form of the Java package "java.awt" in the first version of Java in 1996. It enabled the creation of GUI applications that ran without modifications on the operating systems supported by Java (Flatscher, 2017).

With Swing, another GUI framework was added in Java 2 in 1998, which was combined in the Java package "javax.swing" (Flatscher, 2017).

JavaFX is intended to simplify the creation of graphical user interfaces (GUI) on all systems supported by Java. Parallel to JavaFX, a scripting language called JavaFX Script was introduced. However, this was dropped again with the introduction of JavaFX 2.0 in 2011. With the release of Java 8, JavaFX was also updated to version 8 (Wikipedia, 2022).

2.5 IntelliJ IDEA

As mentioned above, the optional software component IntelliJ IDEA and the associated ooRexx plugin are used in this work.

IntelliJ IDEA is an integrated development environment (IDE) that was first published by JetBrains in 2001. IntelliJ IDEA can be used to develop software written in JVM-based languages (Wikipedia, 2023-a). IntelliJ IDEA is offered in a limited free Community Edition, as well as a paid Ultimate Edition with additional functionality (Wikipedia, 2023-a).

Through the ooRexx Plugin for IntelliJ it is possible to use ooRexx with IntelliJ. Furthermore, the plugin offers the possibility to detect errors during programming and to colouring the syntax of the code.

3 Installation Guide

In the following chapter, installation instructions are given for the required software components, whereby the first three are absolutely necessary and the last two can be installed optionally.

Care should be taken to ensure that the Java and ooRexx programming languages are installed in any case before BSF4ooRexx. Once this has been done, the optional components can be installed in the form of IntelliJ and the associated ooRexx Plugin for IntelliJ. This also corresponds to the sequence described in this manual.

The following software versions were used in this paper:

- Java: Oracle Version 8u351 64 Bit
- ooRexx: Open Object Rexx Version 5.0.0-12531 64 Bit
- BSF4ooRexx: V850-20221106 64 Bit
- IntelliJ IDEA: Version 2022.3
- ooRexx Plugin for IntelliJ IDEA: Version 2.2.0-GA

3.1 Java

First of all, the programming language Java must be installed on the computer if it is not already present. The bit rate should correspond to that of ooRexx. For this thesis, as already mentioned above, version 8 with a bit rate of 64-bit from Oracle was used. The installation file can be found here:

https://www.java.com/de/download/

If an older version of Java is already present on the computer, it can be removed in the course of the installation process, as an uninstall tool is included in the installation programme.

When using Oracle's version, however, make sure that it is only licensed for personal (non-commercial) use, as can be seen in the left image of Figure 1.

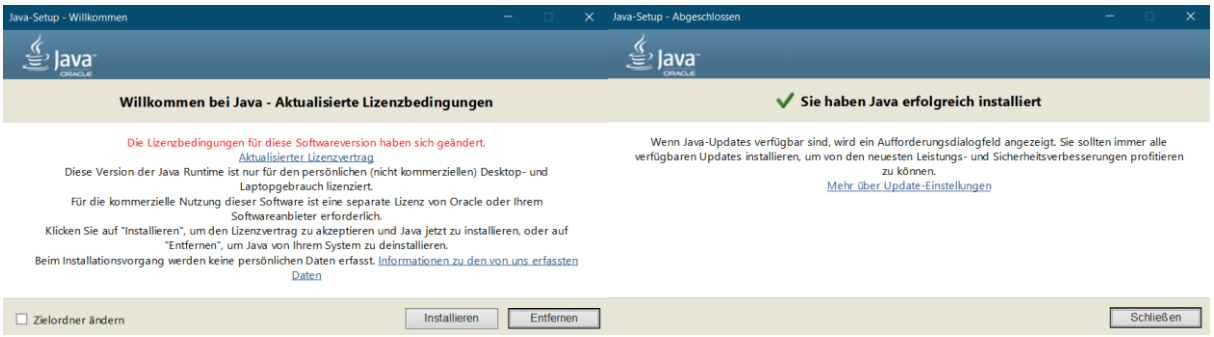

Figure 1: Installation Process Java

3.2 ooRexx 5.0.0

The next step is to install Open Object Rexx on the operating system. It is recommended to use the same bit rate that is used for the operating system and for Java to avoid errors. ooRexx 5.0.0 can be downloaded from the following link:

https://sourceforge.net/projects/oorexx/files/oorexx/

If an older version of ooRexx is already installed on the system, it should be uninstalled first. This applies in particular if the recent version does not have the same bit rate as the previous version. Otherwise, the installer will recognise the previous version and automatically start the uninstall programme.

Figure 2 shows some of the steps that are followed during the installation of ooRexx. During the installation it is not necessary to make any changes to the options.

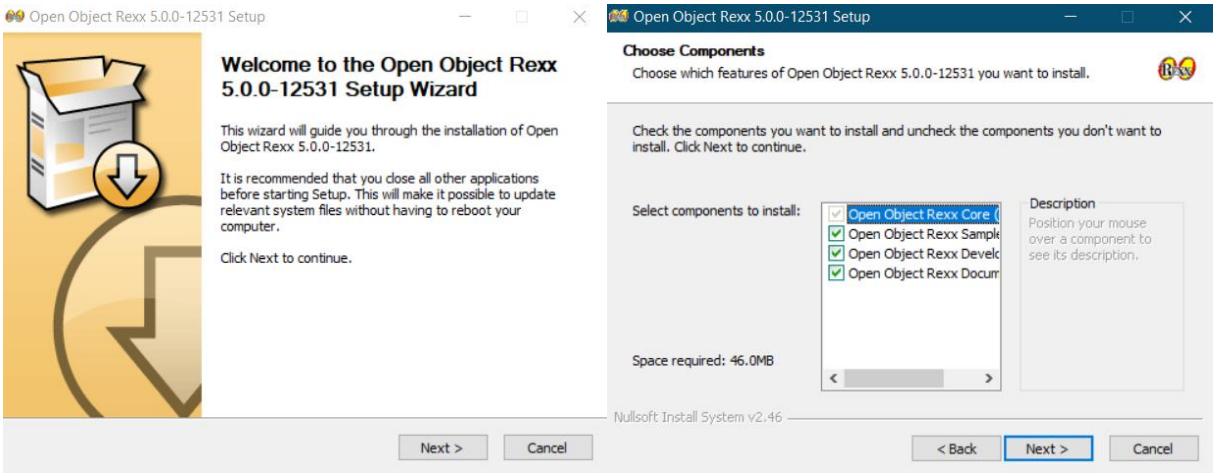

Figure 2: Installation Process ooRexx

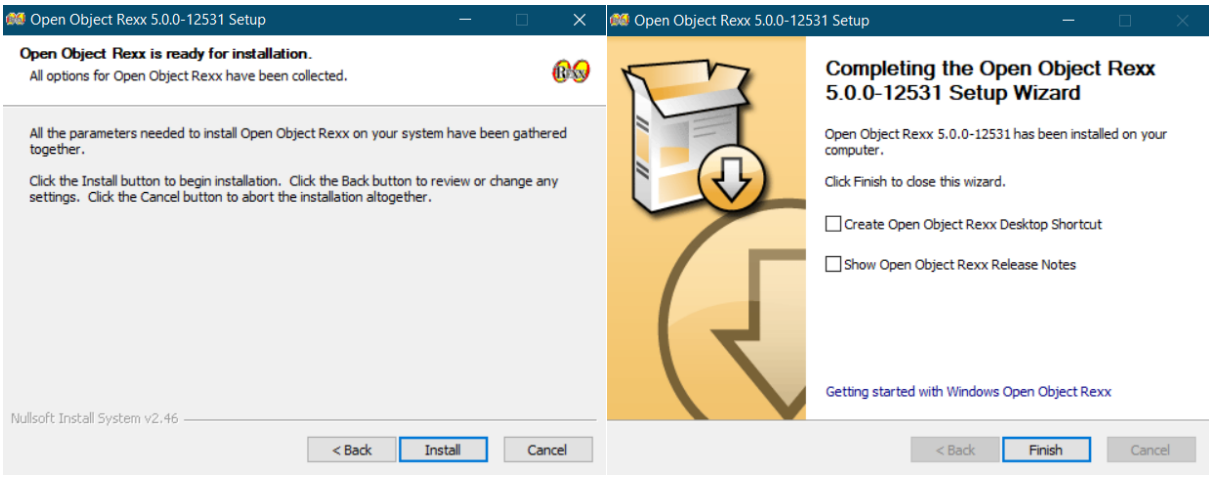

Figure 2 (Continued): Installation Process ooRexx

3.3 BSF4ooRexx850

After successfully installing the two programming languages Java and ooRexx, the Bean Scripting Framework for Open Object Rexx can be downloaded from the SourceForge website under the following link:

https://sourceforge.net/projects/bsf4oorexx/files/

If an older version of BSF4ooRexx already exists, it must be uninstalled before installing the latest version.

The downloaded file is a .zip file, which needs to be unzipped before being installed. It includes the installation files for the different operating systems.

To start the installation process, open the unzipped folder of BSF4ooRexx850, go to "install/windows" and start "install.cmd". This opens a command line window in which the installation is conducted (see Figure 3).

BSF4ooRexx850 automatically detects which architecture is required during the installation. In this case it is a 64-bit version.

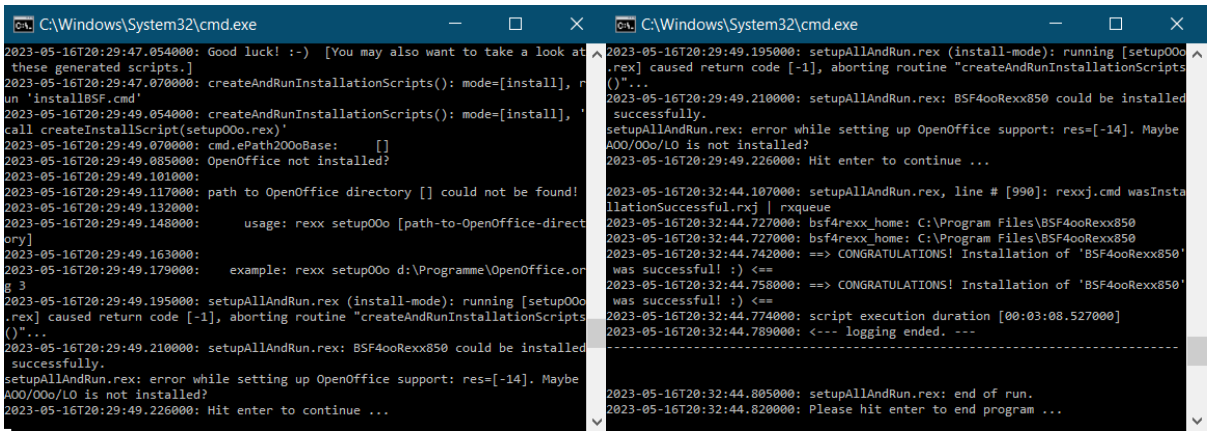

Figure 3: Installation Process BSF4ooRexx

3.4 IntelliJ IDEA

Next, you can install the optional software component IntelliJ. IntelliJ is available in two versions, a paid Ultimate Edition and a free Community Edition. The free version is sufficient for this thesis (see top left image in Figure 4). This can be found under the link below:

https://www.jetbrains.com/idea/download/#section=windows

After starting the installation process, various settings can be made, but these are not necessary for this paper (see top right image in Figure 4).

After completing the installation of IntelliJ, the corresponding ooRexx Plugin for IntelliJ can be installed, which is described in the next point.

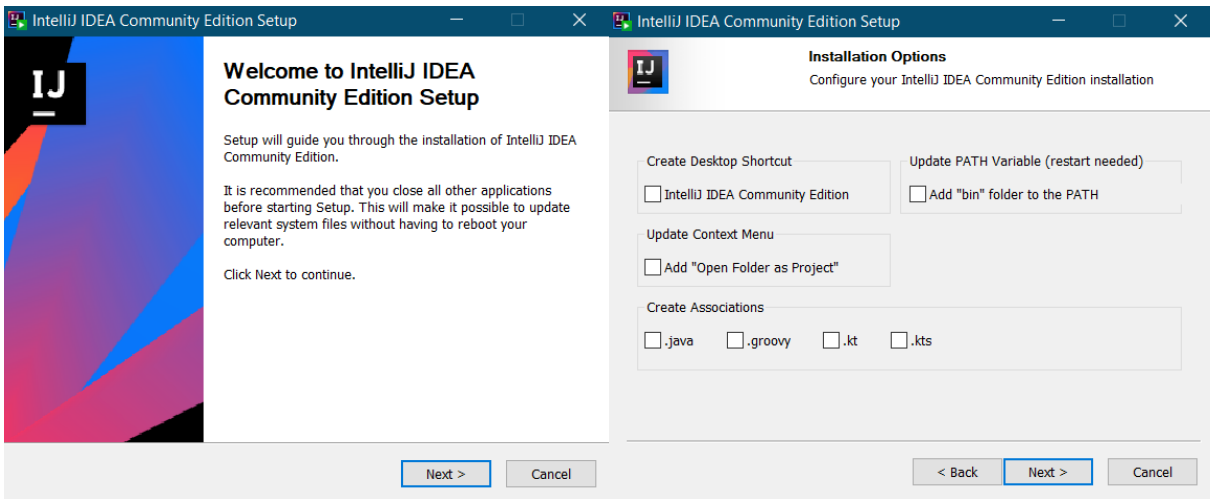

Figure 4: Installation Process IntelliJ

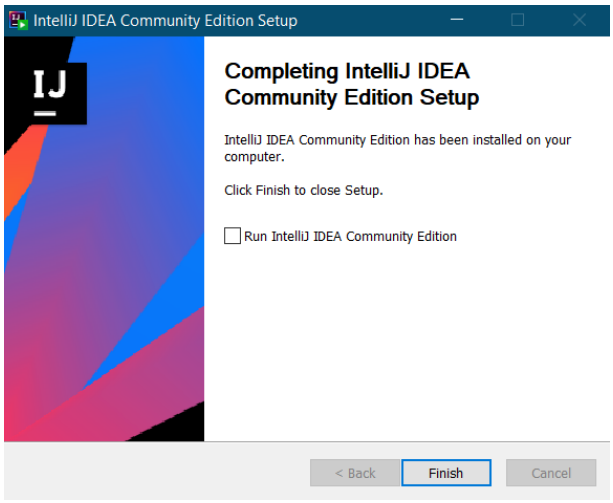

Figure 4 (Continued): Installation Process IntelliJ

3.5 ooRexx Plugin for IntelliJ IDEA

Finally, the ooRexx Plugin for IntelliJ must be installed. This can be downloaded from the following link:

https://sourceforge.net/projects/bsf4oorexx/files/Sandbox/aseik/ooRexxIDEA/GA/

Make sure that the file is not unpacked after the download. This archive is imported by IntelliJ.

Once you have opened IntelliJ, go to the Plugins section. There, first click on the gear icon and then on Install Plugin from Disk (see left image in Figure 5). Select the downloaded ooRexx plugin.

To finish, IntelliJ must be restarted (see right image in Figure 5).

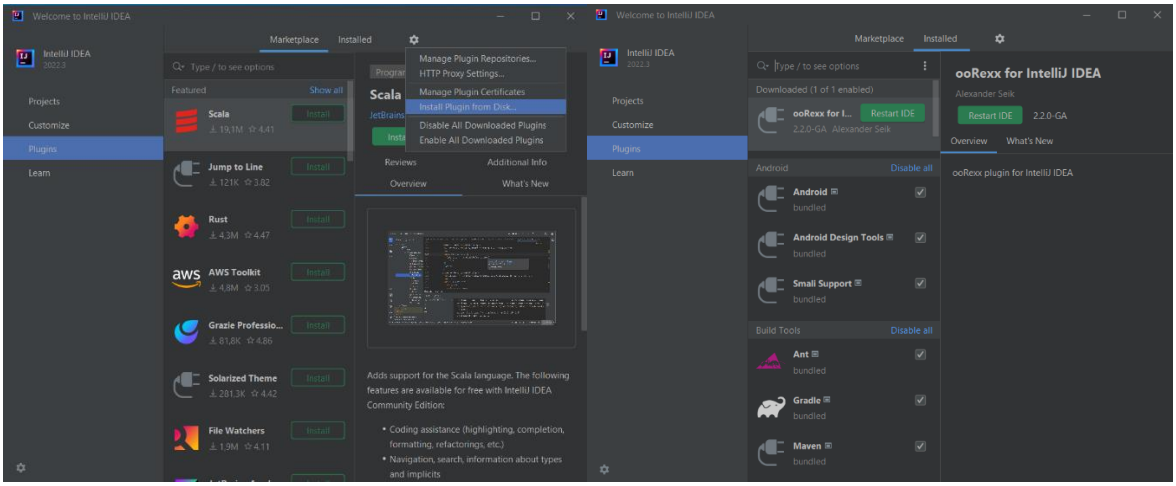

Figure 5: Installation Process ooRexx Plugin

4 Nutshell Examples

This chapter presents the programmed nutshell examples that give an insight into the basics of 3D programming with ooRexx and JavaFX. For each example, the code or a part of it, if it does not differ too much from the previous one, is shown. This code, which is not shown in full, can be found in the appendix. In addition, one or more images of the output of the executed code are provided. The examples should be executed in the order given in this paper, as some examples build on the previous ones. For this paper, the tutorials from https://www.tutorialspoint.com/javafx/index.htm were partly used as inspiration.

4.1 Apache Version 2.0 License

In each of the programmed examples, the Apache License Version 2.0 is specified (see Figure 6). This is a free software licence from the Apache Software Foundation. Under this licence, the software may be freely used, modified and distributed in any environment (Apache Software Foundation, 2004).

```
/*
---------------------- Apache Version 2.0 License ----------------------
Copyright 2023 René Steger

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
------------------------------------------------------------------------
*/
```

Figure 6: Apache Version 2.0 License

4.2 Start Code for each Programme

The start code given in Listing 1 is the code that is executed at the beginning of each example (Flatscher, 2023).

```
19 -- create an instance of the Rexx class
20 rexxHandler=.RexxAppHandler~new
21 -- the abstract "start" method will be served by rexxHandler
22 rxApp=BSFCreateRexxProxy(rexxHandler, ,"javafx.application.Application")
23 -- launch the application and invokes "start"
24 rxApp~launch(rxApp~getClass, .nil)
25
26 -- get Java Support
27 ::REQUIRES "BSF.CLS"
28
29 -- the Rexx handler for javafx.application.Application
30 ::CLASS RexxAppHandler
31
32 -- will be called by JavaFX
33 ::METHOD start
34 -- accesses the primary stage
35 USE ARG primaryStage
```

Listing 1: Start Code for each Programme

First of all, access from ooRexx to Java must be established. The directive "::REQUIRES" is used for this, as can be seen in line 27 of the code. It loads the ooRexx module "BSF.CLS", which masks Java as ooRexx. Thus, all of Java's functionality is present. The directive "::REQUIRES" is always executed before all other non-directive statements.

In line 30, the directive "::CLASS" creates the class called "RexxAppHandler" in ooRexx. An instance of this created class is created in line 20.

To create the initial graphical user interface in JavaFX, the abstract class "javafx.application.Application" and the "start" method must be implemented. The code in line 24 starts the application and calls the start process. In order to create a Java object from a Rexx object, the external Rexx function "BSFCreateRexxProxy()" is required (Flatscher, 2017).

4.3 3D Shapes

3D shapes are generally geometric figures that can be drawn in an XYZ plane. To create such in JavaFX, there are the classes “Sphere”, “Cylinder” and “Box”. These classes just mentioned belong to the Java package “javafx.scene.Shape” (Oracle, n.d.-j).

4.3.1 Create a Sphere

In this example, a three-dimensional sphere is created by the class “Sphere” with a certain size.

```
37 -- setting title to the Stage
38 primaryStage~setTitle("Sphere")
39
40 -- create the root node
41 root=.bsf~new("javafx.scene.Group")
42
43 -- create a sphere
44 sphere=.bsf~new("javafx.scene.shape.Sphere")
45 -- set the radius of the sphere
46 sphere~radius=75
47 -- set position of the sphere
48 sphere~translateX=200
49 sphere~translateY=150
50
51 -- add the object to the root node
52 root~getChildren~~add(sphere)
53
54 -- create a scene and put it on the stage
55 primaryStage~setScene(.bsf~new("javafx.scene.Scene", root, 400, 300))
56 -- display the content of the stage
57 primaryStage~show
```

Listing 2: Create a Sphere

First of all, a title is added to the application. Then a new group is created with “.bsf~new” and defined as the root node. In line 44, a sphere is created by calling the class “Sphere”, which is finally added to the root node in line 52 with “add(sphere)”. With “~radius” the size of the sphere is defined. Since the zero point of the coordinate system in JavaFX is in the upper left corner, the sphere is created there (see left image in Figure 7).Therefore, the next step is to set the position of the sphere, which is done in lines 48 & 49. This moves the sphere in the desired direction (see right image in Figure7). Finally, a scene of the desired size is created and added

to the stage of the application. The last line “primaryStage~show” enables the output of the example.

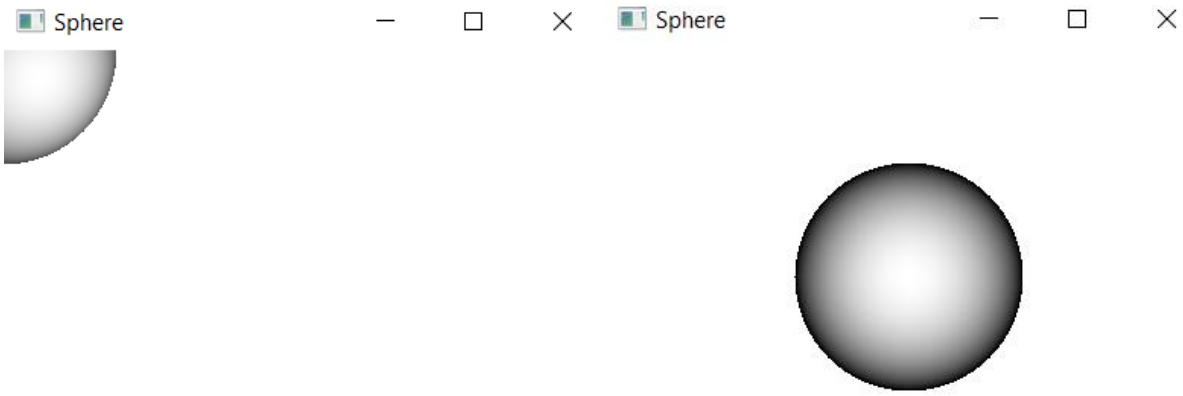

Figure 7: Output from Create a Sphere

4.3.2 Create a Cylinder

In the following example, a three-dimensional cylinder is created with the class “Cylinder” in a certain size.

```
37 -- setting title to the Stage
38 primaryStage~setTitle("Cylinder")
39
40 -- create the root node
41 root=.bsf~new("javafx.scene.Group")
42
43 -- create a cylinder
44 cylinder=.bsf~new("javafx.scene.shape.Cylinder")
45 -- set the radius and the height of the cylinder
46 cylinder~radius=50
47 cylinder~height=200
48 -- set position of the cylinder
49 cylinder~translateX=200
50 cylinder~translateY=150
51
52 -- add the object to the root node
53 root~getChildren~~add(cylinder)
54
55 -- create a scene and put it on the stage
56 primaryStage~setScene(.bsf~new("javafx.scene.Scene", root, 400, 300))
```

```
57 -- display the content of the stage
58 primaryStage~show
```

Listing 3: Create a Cylinder

Since the creation of a cylinder is basically the same as the creation of the sphere in the previous example, we will only briefly go into how the size of the cylinder is determined. First, as can be seen in line 44 of Listing 3, the cylinder must be created with the class “Cylinder”. Then the size of the cylinder is defined by “~radius” and “~height”. The rest of the code corresponds to the previous example and you get the following output (see right picture of Figure 8).

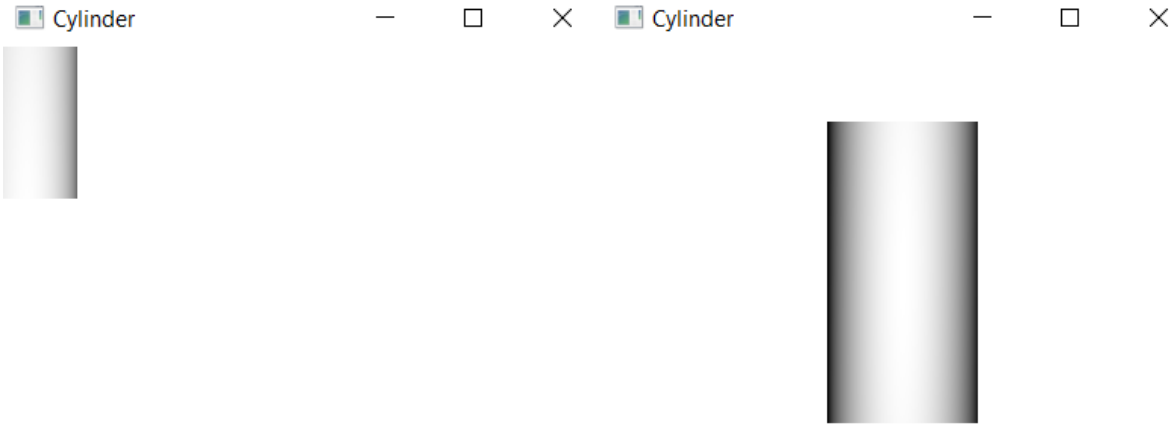

Figure 8: Output from Create a Cylinder

4.3.3 Create a Box

This example is intended to show how to create a three-dimensional box of a certain size using the “Box” class.

```
37 -- setting title to the Stage
38 primaryStage~setTitle("Box")
39
40 -- get access to the JavaFX colors
41 col=bsf.loadClass("javafx.scene.paint.Color")
42
43 -- create the root node
44 root=.bsf~new("javafx.scene.Group")
45
46 -- create a box
47 box=.bsf~new("javafx.scene.shape.Box")
```

```
48 -- set the width, height and depth of the box
49 box~width=150
50 box~height=150
51 box~depth=150
52 -- set position of the box
53 box~translateX=200
54 box~translateY=150
55
56 -- add the object to the root node
57 root~getChildren~~add(box)
58
59 -- create a scene and put it on the stage
60 primaryStage~setScene(.bsf~new("javafx.scene.Scene", root, 400, 300, col~LIGHTGREY))
61 -- display the content of the stage
62 primaryStage~show
```

Listing 4: Create a Box

Creating a box works in the same way as creating a sphere or a cylinder. The only difference to the previous example is that instead of the radius, the width is set with "box~width" and the depth "box~depth" after the box has been created. Through "box=.bsf~new("javafx.scene.shape.Box")" this is possible.

In addition, in this example the background of the scene was changed by "col~LIGHTGREY". However, as can be seen in line 41 of Listing 4, the Java class "javafx.scene.paint.Color" must first be loaded into ooRexx so that it can be accessed (Oracle, n.d.-a). At the same time, the newly created instance of the class has been given the name "col". The result of the code is shown in the right image of Figure 9.

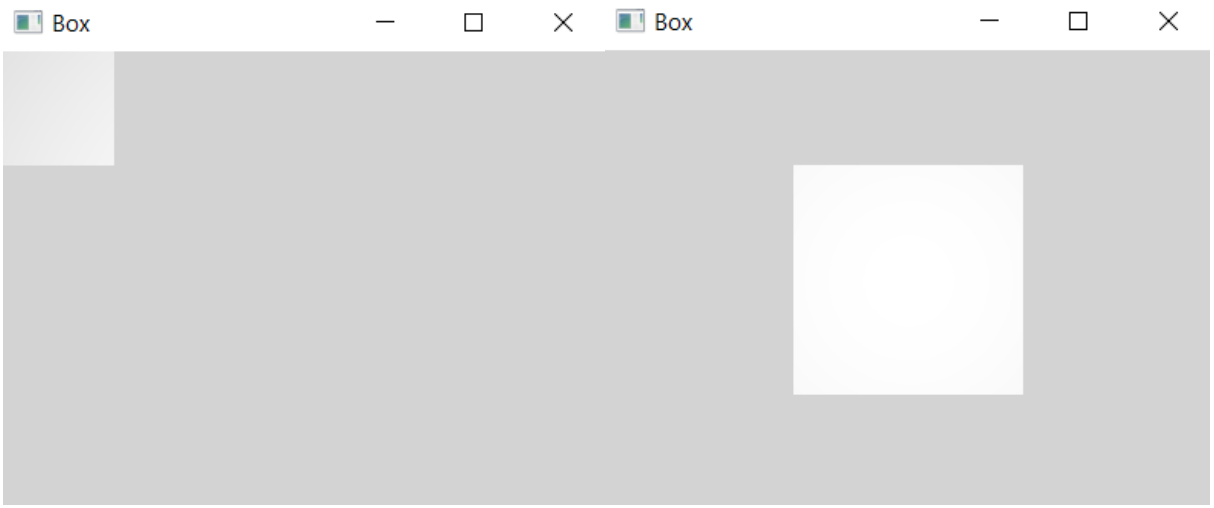

Figure 9: Output from Create a Box

4.4 Create a Perspective Camera

In the last three examples we saw how to create the different 3D shapes. The following examples show how a perspective camera can be added to these. The “PerspectiveCamera” is a subclass of the “Camera” class. The defined position of the camera is always in the middle of a scene (Oracle, n.d.-g).

4.4.1 Box with Perspective Camera

In this example, as mentioned before, a perspective camera is added to a box. Most of the code in the example is the same as in the previous example. The application is given a title, the box is created and its position in the scene is set.

```
37 -- setting title to the Stage
38 primaryStage~setTitle("Box_PerspectiveCamera")
39
40 -- get access to the JavaFX colors
41 col=bsf.loadClass("javafx.scene.paint.Color")
42
43 -- create the root node
44 root=.bsf~new("javafx.scene.Group")
45
46 -- create a box
47 box=.bsf~new("javafx.scene.shape.Box")
48 -- set the width, height and depth of the box
49 box~width=150
50 box~height=150
51 box~depth=150
52 -- set position of the box
53 box~translateX=150
54 box~translateY=150
55
56 -- add the object to the root node
57 root~getChildren~~add(box)
58
59 -- create a perspective camera
60 camera=.bsf~new("javafx.scene.PerspectiveCamera")
61 -- set position of the camera
62 camera~translateX=0
63 camera~translateY=0
```

```
64 camera~translateZ=0
65
66 -- create a scene
67 scene=.bsf~new("javafx.scene.Scene", root, 800, 600, col~LIGHTGREY)
68 -- add camera to the scene
69 scene~setCamera(camera)
70
71 -- put the scene on the stage
72 primaryStage~setScene(scene)
73 -- display the content of the stage
74 primaryStage~show
```

Listing 5: Create a Perspective Camera

Finally, in line 60 of the code, the camera is created. For this, the Java class "javafx.scene.PerspectiveCamera" is imported into ooRexx with the instruction ".bsf~new" and an instance named "camera" is created from it. With "~translate" for the X, Y and Z axis of the coordinate system, the camera position could be moved to the desired location. However, this was not done in the first step. This means that the camera, after being added to the scene in line 69 with "scene~setCamera(camera)", is in the middle of the scene.

The output of the example can be seen in the left image of Figure 10.

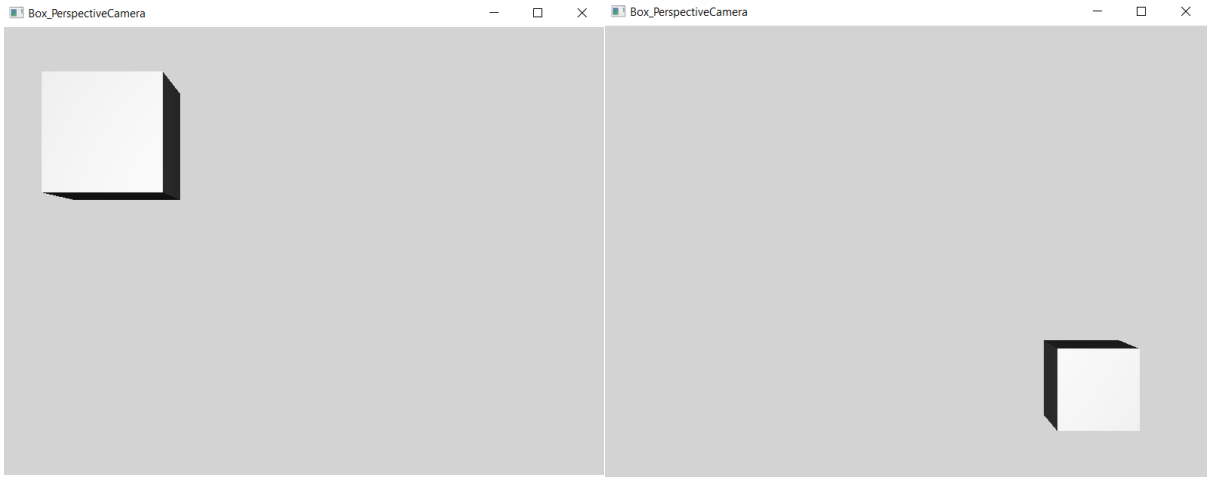

Figure 10: Output from Create a Perspective Camera

Finally, in this short section of the above example, the camera position was changed (see Listing 6). The perspective camera was moved to the left on the X-axis, up on the Y-axis and out of the screen on the Z-axis, resulting in the output shown in Figure 10 on the right.

```
59 -- create a perspective camera
60 camera=.bsf~new("javafx.scene.PerspectiveCamera")
61 -- set position of the camera
62 camera~translateX=-600
63 camera~translateY=-400
64 camera~translateZ=-500
```

Listing 6: Set position of the Perspective Camera

4.4.2 Cylinder & Sphere with Perspective Camera

For these two examples, the programmed code can be found in the appendix, as it is essentially the same as for the previous example with the box. In the left-hand image of Figure 11, the three-dimensionality of the cylinder is clearly visible. The sphere in the right-hand image of Figure 11 is also shown in a weakened form, as it is only slightly distorted. But since it is a sphere, this is to be expected.

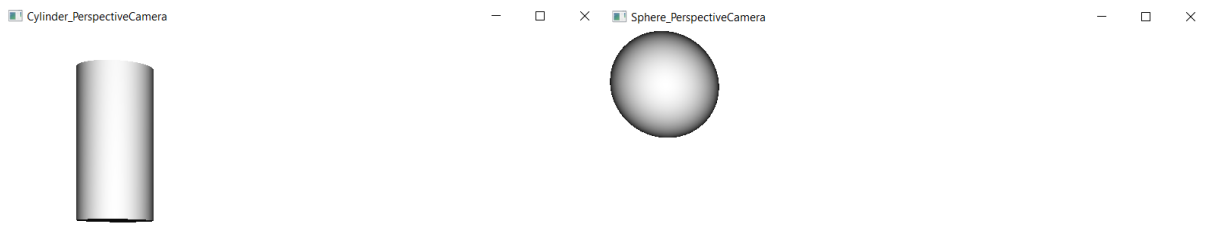

Figure 11: Output from Cylinder & Sphere with Perspective Camera

4.5 Transformations

In the following two examples, the transformation possibilities of the class “Node” are presented, which include scaling, rotating, but also the translating used in the previous examples (Oracle, n.d.-f). The third example shows another possibility to use these transformations with the class “Transform”. With this class, the transformation Shear can also be used (Oracle, n.d.-k).

4.5.1 Scale

The transformation Scale determines the factors with which the coordinates around the centre of the object are scaled along the various axes (Oracle, n.d.-f). In this example, as before, only excerpts of the code with the respective changed values are shown, as there are only three additional lines involved. The entire code can again be found in the appendix.

Listing 7 shows four different factor definitions along the respective axes. Figure 12 shows the corresponding outputs, with the top left image belonging to the top code.

```
52 -- add a scale transformation to the box
53 box~scaleX=1.75
54 box~scaleY=0.75
55 box~scaleZ=0.5
```

```
52 -- add a scale transformation to the box
53 box~scaleX=0
54 box~scaleY=1
55 box~scaleZ=1
```

```
52 -- add a scale transformation to the box
53 box~scaleX=1
54 box~scaleY=0
55 box~scaleZ=1
```

```
52 -- add a scale transformation to the box
53 box~scaleX=1
54 box~scaleY=1
55 box~scaleZ=0
```

Listing 7: Scale Transformation

As can be seen in the top left image, the box from Figure 10 has been given a different shape by adding scaling. A multiplier of 1.75 was applied to the X-axis, 0.75 to the Y-axis and 0.5 to the Z-axis, making the box wider, lower and less deep.

For the other three outputs, a multiplier of 0 was applied to each of the three axes, while a multiplier of 1 was applied to the remaining values. The result is a square and not a box.

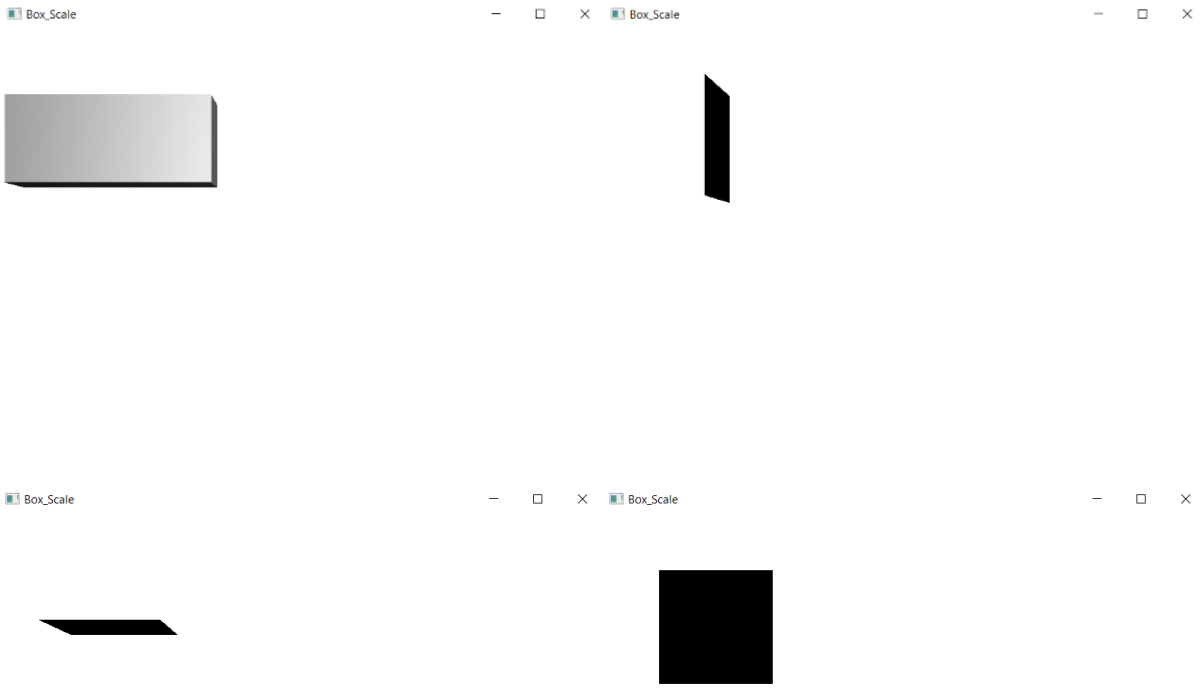

Figure 12: Output from Scale Transformation

4.5.2 Rotate

This example shows the result of adding a rotation to the box known from the above examples. Here too, the entire code can be found in the appendix.

```
52 -- set rotation of the box
53 point=.bsf~new("javafx.geometry.Point3D", 1, 1, 1)
54 box~rotate=45
55 box~rotationAxis=point
```

Listing 8: Set Rotation

First, the class “Point3D” must be imported into ooRexx with ".bsf~new" for the execution of the rotation. At the same time, an instance of this class is created, which is given the name "point". The coordinates X, Y and Z are represented by this geometric point (Oracle, n.d.-i).

With "~rotate" the angle of rotation is set. In this case to 45 degrees. By assigning the point created earlier to the instruction "~rotationAxis", the rotation is executed over all three axes

(see left image in Figure 13). If this instruction is not executed, the box will only rotate around one axis (Oracle, n.d.-f).

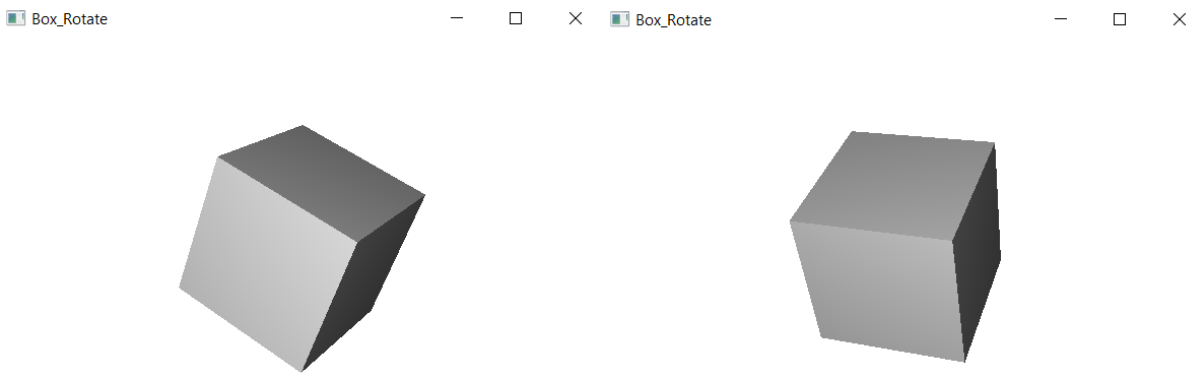

Figure 13: Output from Set Rotation

In Listing 9, different values for the coordinates are passed to the point, triggering a different rotation of the box (see right image in Figure 13).

```
52 -- set rotation of the box
53 point=.bsf~new("javafx.geometry.Point3D", 2, 1, 0)
54 box~rotate=45
55 box~rotationAxis=point
```

Listing 9: Adjusting the Rotation

4.5.3 Translate, Shear, Scale & Rotate with the Class “Transform”

In the following example, the four different transformations are combined. In addition, a different class is used than in the previous examples. At the beginning, the title of the application is defined as usual. Then the box is created with the class “Box” and its size is defined. In this case, the values for width, height and depth are 150.

```
37 -- setting title to the Stage
38 primaryStage~setTitle("Box_Transform")
39
40 -- get access to the JavaFX colors
41 col=bsf.loadClass("javafx.scene.paint.Color")
42
43 -- create the root node
44 root=.bsf~new("javafx.scene.Group")
45
```

```
46 -- create a box
47 box=.bsf~new("javafx.scene.shape.Box")
48 -- set the width, height and depth of the box
49 box~width=150
50 box~height=150
51 box~depth=150
52
53 -- set position of the box
54 translate=.bsf~new("javafx.scene.transform.Translate")
55 translate~x=400
56 translate~y=300
57 translate~z=0
58
59 -- add a shear transformation to the box
60 shear=.bsf~new("javafx.scene.transform.Shear")
61 shear~x=1.75
62 shear~y=1.25
63
64 -- add a scale transformation to the box
65 scale=.bsf~new("javafx.scene.transform.Scale")
66 scale~x=1.75
67 scale~y=0.75
68 scale~z=0.5
69
70 -- set rotation of the box
71 rotate=.bsf~new("javafx.scene.transform.Rotate")
72 point=.bsf~new("javafx.geometry.Point3D", 1, 1, 1)
73 rotate~angle=45
74 rotate~axis=point
75
76 -- add the transformations to the box
77 box~getTransforms~~add(translate)~~add(shear)~~add(scale)~~add(rotate)
78
79 -- add the object to the root node
80 root~getChildren~~add(box)
81
82 -- create a scene and put it on the stage
83 primaryStage~setScene(.bsf~new("javafx.scene.Scene", root, 800, 600, col~LIGHTGREY))
84 -- display the content of the stage
85 primaryStage~show
```

Listing 10: Translate, Shear, Scale and Rotate

Then the subclass "Translate" of the class "Transform" is imported into ooRexx with the ".bsf~new" already known from the previous examples and the instance "translate" is created. The distances by which the coordinates are shifted in the direction of the respective axes are determined with x, y and z. In this case, the box is in the middle of the window.

Next, the class "Shear" is imported and an instance of it called "shear" is created. Shearing is a transformation that was not used in the previous examples, as this is only possible in the class "Transform". Here, a multiplier is defined by which the coordinates of one axis are shifted depending on the other axis. However, this is only possible for x and y. In this example, the multiplier 1.75 was used for x and 1.25 for y.

In the next step, the transformation Scale known from above is added. This time, however, it comes from the class "Transform". It determines the factor by which the coordinates are scaled in the direction of the axes. Here the values 1.75 for x, 0.75 for y and 0.5 for z were applied.

The last step is to add the rotation to the previous steps. This is done as in the example above, where the rotation of an object was introduced. This time, however, the class "Rotate" must be imported into ooRexx as shown in line 71 of Listing 10. Once this has been done, the angle of the rotation, here 45 degrees, is specified.

Finally, the instances "translate", "shear", "scale" and "rotate" are added to the box with "~add()" (see line 77). The box must again be assigned to the root node, otherwise the box will not be visible in the output.

The output of the individual steps is shown in Figure 14 starting from the top left to the bottom right.

Figure 14: Output from Translate, Shear, Scale and Rotate

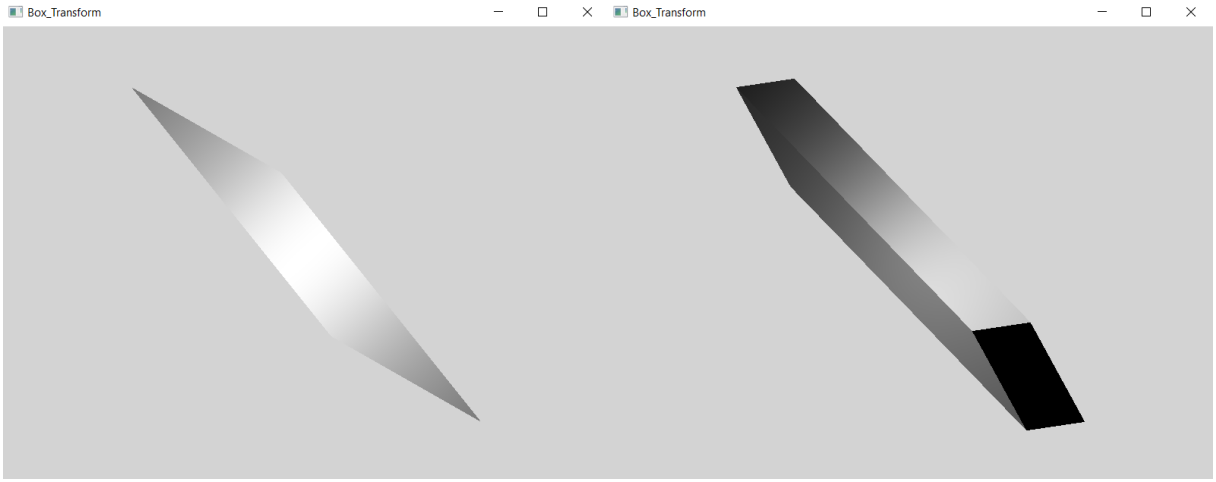

Figure 14 (Continued): Output from Translate, Shear, Scale and Rotate

4.6 CullFace

The following example discusses cullFace in JavaFX. CullFace is a property that can be applied to all three-dimensional shapes. Here it is illustrated using spheres. In general, culling removes the parts of a shape that are not visible in the display area. This can improve the performance of the display (Wikipedia, 2021-b). There are three types of culling, namely "FRONT", "BACK" and "NONE". These are enum constants, i.e. predefined constants that cannot be changed (Oracle, n.d.-b).

```
37 -- setting title to the Stage
38 primaryStage~setTitle("Sphere_CullFace")
39
40 -- import class Sphere
41 sphere=bsf.import("javafx.scene.shape.Sphere")
42 -- get access to cullFace
43 cuFa=bsf.loadClass("javafx.scene.shape.CullFace")
44
45 -- create the root node
46 root=.bsf~new("javafx.scene.Group")
47
48 -- create a sphere
49 sphere1=sphere~new
50 -- set the radius of the sphere
51 sphere1~radius=75
52 -- set position of the sphere
53 sphere1~translateX=150
54 sphere1~translateY=200
```

```
55 -- set the value of cullFace
56 sphere1~cullFace=cuFa~front
57
58 sphere2=sphere~new
59 sphere2~radius=75
60 sphere2~translateX=400
61 sphere2~translateY=200
62 sphere2~cullFace=cuFa~back
63
64 sphere3=sphere~new
65 sphere3~radius=75
66 sphere3~translateX=650
67 sphere3~translateY=200
68 sphere3~cullFace=cuFa~none
69
70 -- add the objects to the root node
71 root~getChildren~~add(sphere1)~~add(sphere2)~~add(sphere3)
72
73 -- create a scene and put it on the stage
74 primaryStage~setScene(.bsf~new("javafx.scene.Scene", root, 800, 400))
75 -- display the content of the stage
76 primaryStage~show
```

Listing 11: CullFace

At the beginning of this example, the title of the application is set and the class "Sphere" is imported into ooRexx with the instruction "bsf.import" not yet used in the previous examples (see line 41 in Listing 11). With "bsf.loadClass", access to “CullFace” in ooRexx is enabled. The instance thus created is given the name "cuFa". In the next step, a group is created and set as the root node.

Next, a total of three spheres ("sphere1", "sphere2", "sphere3") with the same size are created in the example with "~new". These are moved to the desired position with "~translate", as they are all created in the upper left corner.

In the next step, the three different culling options are applied to one sphere each (see lines 56, 62 & 68). This is done by assigning this property to the spheres with "~cullFace". With "=cuFa~” the desired value is passed. For "sphere1" this is the enum constant "FRONT", for "sphere2" "BACK" and for "sphere3" "NONE".

This has the effect that for "FRONT" all polygons at the front of the sphere are removed. The standard representation of an object can be seen in "sphere2" with the value "BACK". In this case, all non-visible polygons on the back of the sphere are removed, as drawing these polygons is unnecessary. "sphere3" represents the output of "NONE". In the case of "NONE", no culling is carried out (see Figure 15).

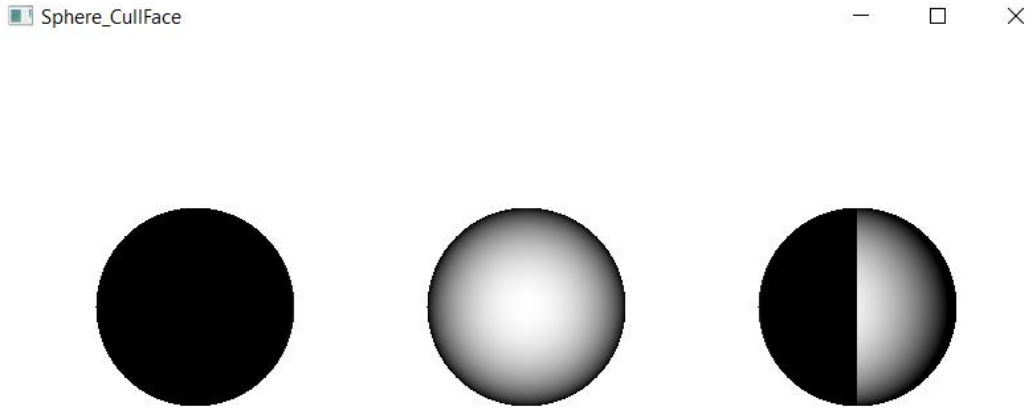

Figure 15: Output from CullFace

4.7 DrawMode

After illustrating in the last example what the cullFace property does, this example introduces the drawMode property. This property can again be applied to all three-dimensional shapes. This time the class "Box" is used for this. There are two forms of drawMode, on the one hand the enum constant "FILL" and on the other hand the enum constant "LINE" (Oracle, n.d.-c).

```
37 -- setting title to the Stage
38 primaryStage~setTitle("Box_DrawMode")
39
40 -- import class Box
41 box=bsf.import("javafx.scene.shape.Box")
42 -- get access to drawMode
43 drMo=bsf.loadClass("javafx.scene.shape.DrawMode")
44 -- get access to the JavaFX colors
45 col=bsf.loadClass("javafx.scene.paint.Color")
46
47 -- create the root node
```

```
root=.bsf~new("javafx.scene.Group")

-- create a box
box1=box~new
-- set the width, height and depth of the box
box1~width=150
box1~height=150
box1~depth=150
-- set position of the box
box1~translateX=200
box1~translateY=150
box1~translateZ=0
-- set the value of drawMode
box1~drawMode=drMo~line

box2=box~new
box2~width=150
box2~height=150
box2~depth=150
box2~translateX=700
box2~translateY=500
box2~translateZ=400
box2~drawMode=drMo~fill

-- add the objects to the root node
root~getChildren~~add(box1)~~add(box2)

-- create a perspective camera
camera=.bsf~new("javafx.scene.PerspectiveCamera")
-- set position of the camera
camera~translateX=0
camera~translateY=0
camera~translateZ=0

-- create a scene
scene=.bsf~new("javafx.scene.Scene", root, 800, 600, col~LIGHTGREY)
-- add camera to the scene
scene~setCamera(camera)

-- put the scene on the stage
primaryStage~setScene(scene)
```

```
89 -- display the content of the stage
90 primaryStage~show
```

Listing 12: DrawMode

The example is similar to the previous example, except that the class "Box" is now imported instead of the class "Sphere". Unlike before, access to drawMode in ooRexx is now enabled in line 43. At the same time, the name of the created instance is defined as "drMo".

In the further course, two boxes are created in this programme. For this purpose, their size is determined and they are moved to the desired position. The position of the perspective camera, which is also created, is set to the centre of the window.

The drawMode property is added to the two boxes with "~drawMode". With "drMo~", a value of this property is assigned to each box. In this case, "box1" is assigned the enum constant "LINE" and "box2" the enum constant "FILL" (see lines 61 & 70 in Listing 12).

Figure 16 shows the output of the two generated boxes. Here you can see that the right box ("box2") is the standard representation of a 3D shape. The left box ("box1"), on the other hand, is drawn with lines only. As these lines are sometimes difficult to recognise on a white background, the colour of the background has also been changed.

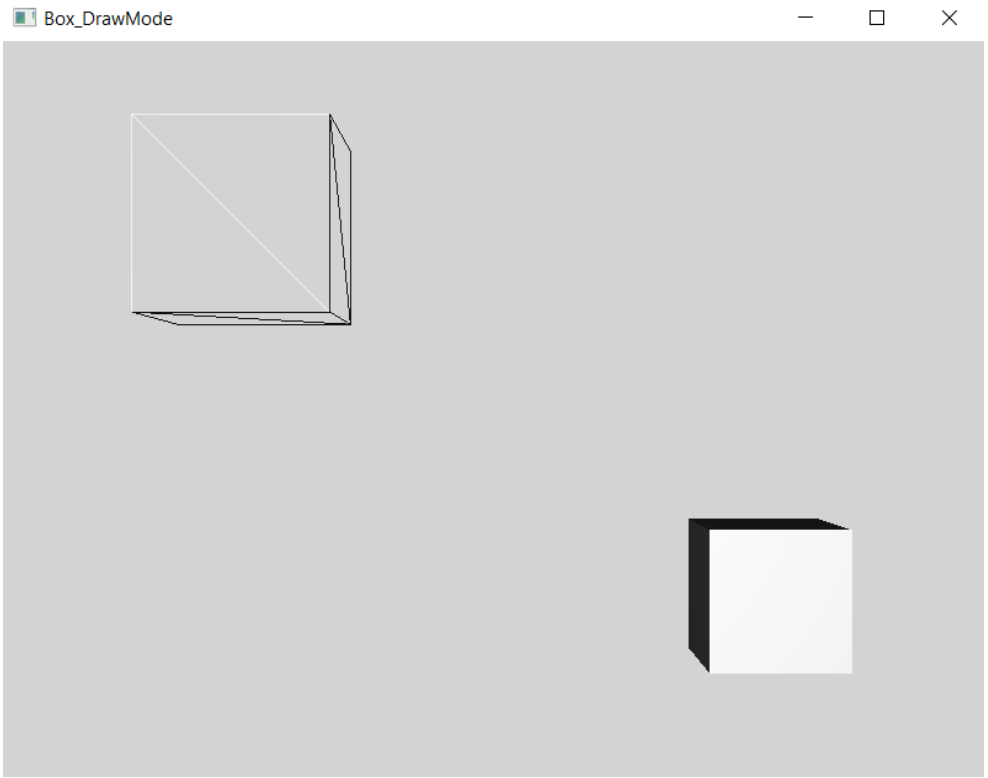

Figure 16: Output from DrawMode

4.8 PhongMaterial

In the following examples, the third property of 3D shapes, namely material, is presented. The subclass "PhongMaterial" of the class "Material" is used to represent the different manifestations of the property Material. The characteristics are diffuseColor, specularColor, specularPower, diffuseMap, bumpMap, specularMap and selfIlluminationMap. These in turn can be added to any three-dimensional object (Oracle, n.d.-h).

4.8.1 DiffuseColor, SpecularColor & SpecularPower

The first example in this section shows the characteristics diffuseColor, specularColor and specularPower of the property Material. DiffuseColor is the basic colour of an object. With specularColor simple reflections are created on an object. With specularPower the intensity of these reflections can be increased. The default value is 32.

```
37 -- setting title to the Stage
38 primaryStage~setTitle("Cylinder_Material")
39
40 -- import class Cylinder
41 cylinder=bsf.import("javafx.scene.shape.Cylinder")
42 -- import class PhongMaterial
43 phMa=bsf.import("javafx.scene.paint.PhongMaterial")
44 -- get access to the JavaFX colors
45 col=bsf.loadClass("javafx.scene.paint.Color")
46
47 -- create the root node
48 root=.bsf~new("javafx.scene.Group")
49
50 -- create a cylinder
51 cylinder1=cylinder~new
52 -- set the radius and the height of the cylinder
53 cylinder1~radius=50
54 cylinder1~height=200
55 -- set position of the cylinder
56 cylinder1~translateX=150
57 cylinder1~translateY=150
58
59 -- preparing the material
60 material1=phMa~new
61 material1~diffuseColor=col~BURLYWOOD
```

```
62
63 -- set the material to cylinder
64 cylinder1~setMaterial(material1)
65
66 cylinder2=cylinder~new
67 cylinder2~radius=50
68 cylinder2~height=200
69 cylinder2~translateX=400
70 cylinder2~translateY=150
71
72 material2=phMa~new
73 material2~specularColor=col~BURLYWOOD
74
75 cylinder2~setMaterial(material2)
76
77 cylinder3=cylinder~new
78 cylinder3~radius=50
79 cylinder3~height=200
80 cylinder3~translateX=650
81 cylinder3~translateY=150
82
83 material3=phMa~new
84 material3~specularPower=10
85
86 cylinder3~setMaterial(material3)
87
88 -- add the objects to the root node
89 root~getChildren~~add(cylinder1)~~add(cylinder2)~~add(cylinder3)
90
91 -- create a perspective camera
92 camera=.bsf~new("javafx.scene.PerspectiveCamera")
93 -- set position of the camera
94 camera~translateX=0
95 camera~translateY=0
96 camera~translateZ=0
97
98 -- create a scene
99 scene=.bsf~new("javafx.scene.Scene", root, 800, 600)
100 -- add camera to the scene
101 scene~setCamera(camera)
102
```

```
103 -- put the scene on the stage
104 primaryStage~setScene(scene)
105 -- display the content of the stage
106 primaryStage~show
```

Listing 13: DiffuseColor, SpecularColor & SpecularPower

At the beginning of the example, the classes "Cylinder" and "PhongMaterial" are imported into ooRexx, as they are needed in the further course of the programme. The instance of the class "Cylinder" is given the name "cylinder". The instance of the class "PhongMaterial" has the name "phMa". In addition, the class "Color" from the package "javafx.scene.paint" must be accessible in ooRexx.

Thus, the three required cylinders can now be created. This is done with the assignment "=cylinder~new". Radius and height of the cylinders as well as their position in the window are determined next.

In the next step, the different types of material are prepared. To do this, "material1", "material2" and "material3" are created with the assignment "=phMa~new". In lines 61,73 & 84 (see Listing13) the respective characteristics are added to the previously created materials. "material1" gets "diffuseColor" as the material surface and "material2" gets "specularColor". These two are assigned the colour "BURLYWOOD" with the instance "col" of the class "Color". The property "specularPower" is applied to "material3".

Now the respective material types must be assigned to the cylinders with "~setMaterial(material)" as shown in lines 64, 75 & 86.

The output of this example is shown in Figure 17.

Figure 17: Output from DiffuseColor, SpecularColor & SpecularPower

4.8.2 DiffuseMap

The second example in this section shows the application of a diffuse map to an object. With this type of material, it is possible to add an image to an object. The supported file formats are "BMP", "GIF", "JPEG" and "PNG" (Oracle, n.d.-d).

The images "wood.jpg", "normal.jpg" and "spec.jpg" used in the following examples and the associated licence conditions can be found under the links in the references (Texturise, 2013-c; 2013-a; 2013-b; n.d.).

Due to the licence conditions, it must be ensured that these or similar images are saved on the computer under the previously specified names before the programmes are executed (Texturise, n.d.).

```
37 -- change directory to program location
38 parse source . . s
39 call directory filespec('loc', s)
40
41 -- setting title to the Stage
42 primaryStage~setTitle("Box_DiffuseMap")
43
```

```
-- create the root node
root=.bsf~new("javafx.scene.Group")

-- create a box
box=.bsf~new("javafx.scene.shape.Box")
-- set the width, height and depth of the box
box~width=150
box~height=150
box~depth=150
-- set position of the box
box~translateX=300
box~translateY=200
-- set rotation of the box
point=.bsf~new("javafx.geometry.Point3D", 1, 1, 1)
box~rotate=45
box~rotationAxis=point

-- load an image
img=.bsf~new("javafx.scene.image.Image", "file:wood.jpg")

-- preparing the material
material=.bsf~new("javafx.scene.paint.PhongMaterial")
material~diffuseMap=img

-- set the material to box
box~setMaterial(material)

-- add the object to the root node
root~getChildren~~add(box)

-- create a perspective camera
camera=.bsf~new("javafx.scene.PerspectiveCamera")
-- set position of the camera
camera~translateX=0
camera~translateY=0
camera~translateZ=0

-- create a scene
scene=.bsf~new("javafx.scene.Scene", root, 600, 400)
-- add camera to the scene
scene~setCamera(camera)
```

```
85
86 -- put the scene on the stage
87 primaryStage~setScene(scene)
88 -- display the content of the stage
89 primaryStage~show
```

Listing 14: DiffuseMap

First, the directory in which the image loaded later in this example is located must be changed to the program location. This is done by "parse source" and "call directory filespec()".

The next lines correspond to many of the examples discussed earlier in the paper. These are lines 42 - 59 from Listing 14, which are therefore not explained again here.

Line 62 again contains code not yet used in this work. Here the class "Image" from the package "javafx.scene.image" is imported into ooRexx and at the same time an instance of the class named "img" is created. Simultaneously, an image is loaded into the programme with "file:". In this case it is a picture called "wood" (Texturise, 2013-c).

Then the class "PhongMaterial" from the package "javafx.scene.paint" is imported into ooRexx. For this purpose, the instance "material" is also created. In line 66, this material type is now added to "material" with "~diffuseMap". The image previously loaded into the programme is next assigned to "material" with "~img".

In line 69, the "material" just created is added to the box created in the programme with "~setMaterial(material)".

The remaining lines of code present in this example are again identical to the examples discussed earlier in the paper.

The output of this example corresponds to that in Figure 18.

Figure 18: Output from DiffuseMap

4.8.3 BumpMap

The third example in this section deals with bump maps. With this type of material, details can be added to 3D shapes without increasing the number of polygons. This is possible because the grey levels of an image are used to create variations in the shading of the surface (Wikipedia, 2021-a). In JavaFX, however, bump maps are essentially normal maps that are stored as RGB images (Oracle, n.d.-h). All the written code for this example is in the appendix.

```
61 -- load an image
62 img=.bsf~new("javafx.scene.image.Image", "file:normal.jpg")
63
64 -- preparing the material
65 material=.bsf~new("javafx.scene.paint.PhongMaterial")
66 material~bumpMap=img
67
68 -- set the material to box
69 box~setMaterial(material)
```

Listing 15: BumpMap

The code for the bump map is largely the same as the example explained earlier. The only two differences are in the area of the image and the material type added to the "material". Instead of the previous image called "wood", this time one called "normal" is loaded into the

programme (Texturise, 2013-a). Furthermore, the “diffuseMap” previously added to the "material" is replaced by a "bumpMap". Figure 19 illustrates the output of the example.

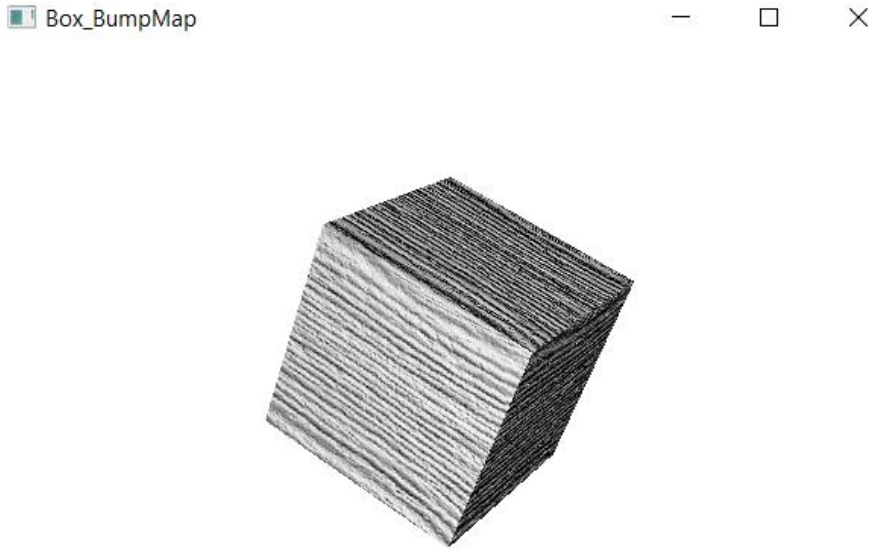

Figure 19: Output from BumpMap

4.8.4 SpecularMap

This example shows the use of a specular map. The purpose of such a map is to define how much an object shines in a certain place. It should be a black and white image added to the object. The whiter a point is on the image added to the object, the more the 3D shape should shine at this point (We Design Virtual, 2020). Here you will also find the complete programme in the appendix.

```
61 -- load an image
62 img=.bsf~new("javafx.scene.image.Image", "file:spec.jpg")
63
64 -- preparing the material
65 material=.bsf~new("javafx.scene.paint.PhongMaterial")
66 material~specularMap=img
67
68 -- set the material to box
69 box~setMaterial(material)
```

Listing 16: SpecularMap

The code in this programme is basically structured like the one in the previous example. This time the last image "normal" is replaced by the image "spec" (Texturise, 2013-b). The second point is to use a "specularMap" instead of the "bumpMap" added to the "material" in the last example. Figure 20 shows the result of the programmed example.

Figure 20: Output from SpecularMap

4.8.5 SelfIlluminationMap

In the following example, we will look at what a self-illumination map does in the programme. A self-illumination map is a type of material that illuminates itself. This means that this area is not affected by other light sources. With such a map, white areas of an added image are completely self-illuminating, whereas the black areas do not illuminate at all (Autodesk, n.d.). The complete example can be found in the appendix.

```
61 -- load an image
62 img=.bsf~new("javafx.scene.image.Image", "file:wood.jpg")
63
64 -- preparing the material
65 material=.bsf~new("javafx.scene.paint.PhongMaterial")
66 material~selfIlluminationMap=img
67
68 -- set the material to box
69 box~setMaterial(material)
```

Listing 17: SelfIlluminationMap

This programme also corresponds to the examples described above. However, the image "wood" is used again in the code and thus loaded into the programme (Texturise, 2013-c). In addition, a "selfIllumiationMap", which is added to the "material", replaces the "specularMap" used previously in the code. The output of the programme can be seen in Figure 21.

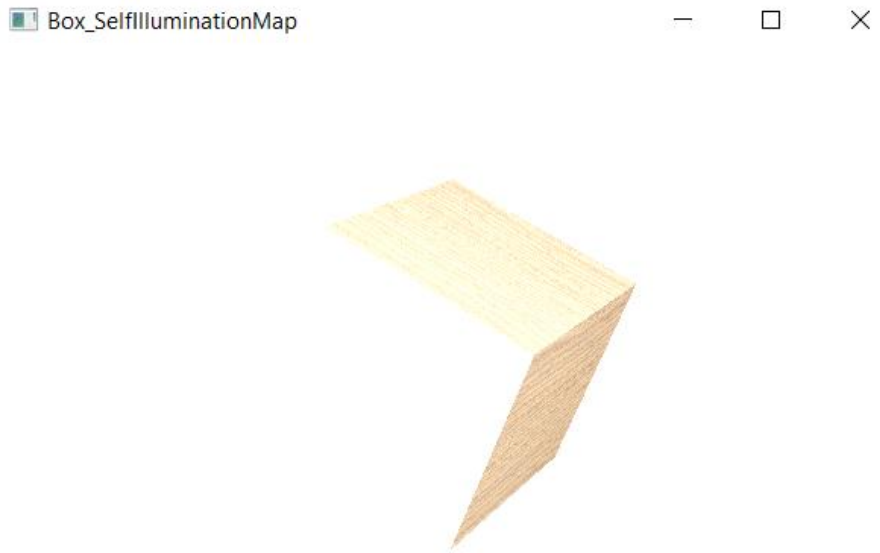

Figure 21: Output from SelfIlluminationMap

4.8.6 DiffuseMap, BumpMap, SpecularMap & SpecularPower

The last example in this section aims to show what output can be generated by a combination of the different material types. For this purpose, the diffuseMap, the bumpMap, the specularMap and specularPower are used. What the result looks like when these are used individually was made clear in the previous examples. Since only a short part of the programme needs to be shown here, the complete code can be found in the appendix.

```
61 -- load images
62 img=.bsf~new("javafx.scene.image.Image", "file:wood.jpg")
63 img2=.bsf~new("javafx.scene.image.Image", "file:normal.jpg")
64 img3=.bsf~new("javafx.scene.image.Image", "file:spec.jpg")
65
66 -- preparing the material
67 material=.bsf~new("javafx.scene.paint.PhongMaterial")
68 material~diffuseMap=img
69 material~bumpMap=img2
70 material~specularMap=img3
```

```
71 material~specularPower=15
72
73 -- set the material to box
74 box~setMaterial(material)
```

Listing 18: DiffuseMap, BumpMap, SpecularMap & SpecularPower

After the code from the example in chapter "4.8.2 DiffuseMap" has been programmed up to line 59, the code in Listing 18 can be continued. The class "Image" from the package "javafx.scene.image" is imported with ".bsf~new" and at the same time the instance "img" is created. The image named "wood" is loaded into this instance (Texturise, 2013-c). This time, however, two more images are loaded into the programme, namely "normal" and "spec" (Texturise, 2013-a; 2013-b). Next, the class "PhongMaterial" is imported into ooRexx and the "material" is created at the same time. Now first the "diffuseMap" is added to the "material" and then "img", i.e. the image "wood" is assigned to it. This gives the box a certain wooden surface. After the "bumpMap" is added to the "material" and the image "img2" is assigned, a certain structure appears on the box. If finally the "specularMap" is added to the "material" and the image "img3" is assigned, and in the last step "specularPower" is set to the value 15 in this case, the result shown in Figure 22 is achieved.

Figure 22: Output from DiffuseMap, BumpMap, SpecularMap & SpecularPower

4.9 LightBase

In the next examples, the class "LightBase" is introduced. This base class consists of the two subclasses "AmbientLight" and "PointLight". It defines properties for objects that represent a form of light source (Oracle, n.d.-e).

4.9.1 AmbientLight

In the first example of the last section, the class "AmbientLight" is explained. In this class, an object is defined as an ambient light source. This means that a light source is created that appears to come from all directions.

```
37 -- change directory to program location
38 parse source . . s
39 call directory filespec('loc', s)
40
41 -- setting title to the Stage
42 primaryStage~setTitle("Box_AmbientLight")
43
44 -- get access to the JavaFX colors
45 col=bsf.loadClass("javafx.scene.paint.Color")
46
47 -- create the root node
48 root=.bsf~new("javafx.scene.Group")
49
50 -- create a box
51 box=.bsf~new("javafx.scene.shape.Box")
52 -- set the width, height and depth of the box
53 box~width=200
54 box~height=200
55 box~depth=200
56 -- set position of the box
57 box~translateX=300
58 box~translateY=200
59 -- set rotation of the box
60 point=.bsf~new("javafx.geometry.Point3D", 1, 1, 1)
61 box~rotate=45
62 box~rotationAxis=point
63
64 -- load an image
65 img=.bsf~new("javafx.scene.image.Image", "file:wood.jpg")
```

```
66
67 -- preparing the material
68 material=.bsf~new("javafx.scene.paint.PhongMaterial")
69 material~diffuseMap=img
70
71 -- set the material to box
72 box~setMaterial(material)
73
74 -- create an ambient light
75 light=.bsf~new("javafx.scene.AmbientLight")
76 light~color=col~CORAL
77
78 -- add objects to the root node
79 root~getChildren~~add(box)~~add(light)
80
81 -- create a perspective camera
82 camera=.bsf~new("javafx.scene.PerspectiveCamera")
83 -- set position of the camera
84 camera~translateX=0
85 camera~translateY=0
86 camera~translateZ=0
87
88 -- create a scene
89 scene=.bsf~new("javafx.scene.Scene", root, 600, 400)
90 -- add camera to the scene
91 scene~setCamera(camera)
92
93 -- put the scene on the stage
94 primaryStage~setScene(scene)
95 -- display the content of the stage
96 primaryStage~show
```

Listing 19: AmbientLight

At the beginning of the example, the directory is changed by "parse source" and "call directory filespec()".

Then the application is given the name "Box_AmbientLight" by "~setTitle" and the class "Color" from the package "javafx.scene.paint" is imported into ooRexx. At the same time, the instance named "col" of this class is created. Then a group is created by ".bsf~new" and this is set as the "root" node.

Next, a box is created by the class "Box" from the package "javafx.scene.shape" and given the name "box". Afterwards, the size of the box is set to 200 by "~width", "~height" and "~depth". Afterwards, the box created in the upper left corner is moved to the centre of the created window by "~translateX" and "~translateY".

In addition, a rotation of the box is added. For this purpose, a geometric point called "point" is created with the class "Point3D" from the package "javafx.geometry". For this point, the values of the coordinates of X, Y and Z are each set to 1.

Then the class "Image" from the package "javafx,scene.image" loads the image "wood" into the programme (Texturise, 2013-c). This image has the name "img" in the code.

Now the class "PhongMaterial" from the package "javafx.scene.paint" creates a new material called "material". In the next step, the material type "diffuseMap" is added to this. The image "img" is then assigned to it. Now the material of the box must be added by "~setMaterial(material)".

Subsequently, the class "AmbientLight" from the package "javafx.scene" creates an ambient light named "light". This is assigned the colour "CORAL" by "col".

Now the box and the light must be added to the "root" node by "add()".

Besides, a camera with the name "camera" is created by the class "PerspectiveCamera" from the package "javafx.scene". Its position is set to the centre of the window by "translateX", "translateY" and "translateZ" with the respective value 0.

Furthermore, the class "Scene" from the package "javafx.scene" creates a scene with the values 600 and 400 for its width and height. In addition, the previously created camera is added to the scene with "~setCamera(camera).

This scene is added to the stage with "~setScene(scene)". Finally, the output of the programmed example is started with "~show".

The result of this programme can be seen in Figure 23.

Figure 23: Output from AmbientLight

4.9.2 PointLight

In this example, the class "PointLight" is briefly discussed. This is a light source object that has a fixed point in a room. It radiates light evenly in all directions. Since in this example only the class "PointLight" is used instead of the class "AmbientLight" and nothing else has changed in the code, only this short excerpt of the code is shown here. The entire programmed code of the example is available in the appendix.

```
74 -- create a point light
75 light=.bsf~new("javafx.scene.PointLight")
76 light~color=col~WHITE
77 -- set position of the light
78 light~translateX=500
79 light~translateY=100
80 light~translateZ=-100
```

Listing 20: PointLight

Up to line 72, the code is the same as in the previous example, so it can be taken over from there.

In this programme, the class "PointLight" from the package "javafx.scene" creates a point light called "light". Then the colour "WHITE" is assigned to it by "col". Then the position of the

light is defined in the room with the values 500, 100 and -100 for "translateX", "translateY" and "translateZ".

The rest of the code can now be taken over again from the previous example. The output of this programme is shown in Figure 24.

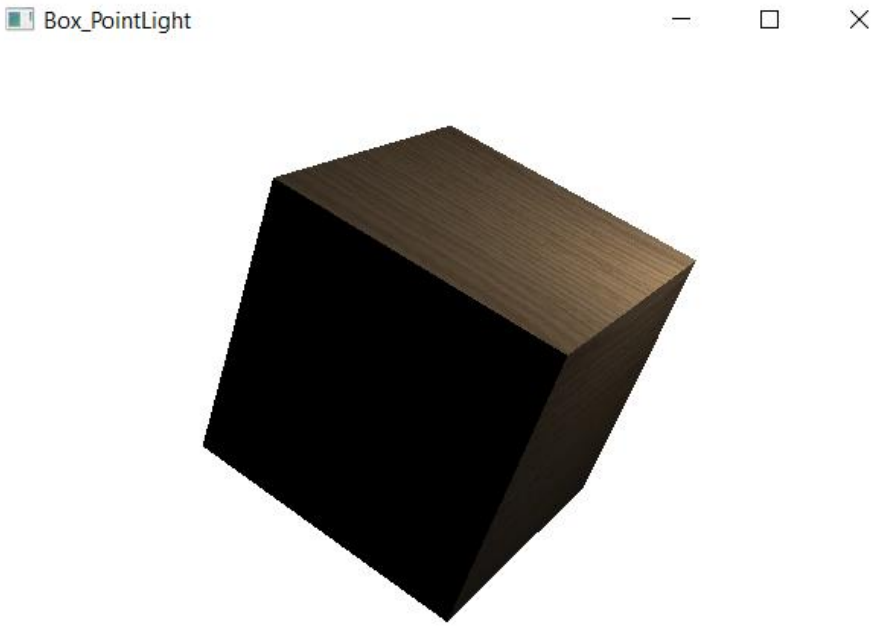

Figure 24: Output from PointLight

4.9.3 AmbientLight & DrawMode

The last example of this thesis shows how the hardly visible edges of an object (see Figure 23 in Chapter 4.9.1) can be displayed more clearly with the help of the class "drawMode", which has already been explained above. The complete code can again be found in the appendix.

```
44 -- import class Box
45 box=bsf.import("javafx.scene.shape.Box")
46 -- get access to drawMode
47 drMo=bsf.loadClass("javafx.scene.shape.DrawMode")
48 -- get access to the JavaFX colors
49 col=bsf.loadClass("javafx.scene.paint.Color")
50
51 -- create the root node
52 root=.bsf~new("javafx.scene.Group")
53
54 -- create a box
55 box1=box~new
```

```
56 -- set the width, height and depth of the box
57 box1~width=200
58 box1~height=200
59 box1~depth=200
60 -- set position of the box
61 box1~translateX=300
62 box1~translateY=200
63 -- set rotation of the box
64 point=.bsf~new("javafx.geometry.Point3D", 1, 1, 1)
65 box1~rotate=45
66 box1~rotationAxis=point
67 -- set the value of drawMode
68 box1~drawMode=drMo~fill
69
70 -- load an image
71 img=.bsf~new("javafx.scene.image.Image", "file:wood.jpg")
72
73 -- preparing the material
74 material=.bsf~new("javafx.scene.paint.PhongMaterial")
75 material~diffuseMap=img
76
77 -- set the material to box
78 box1~setMaterial(material)
79
80 -- create an ambient light
81 light=.bsf~new("javafx.scene.AmbientLight")
82 light~color=col~CORAL
83
84 box2=box~new
85 box2~width=200
86 box2~height=200
87 box2~depth=200
88 box2~translateX=300
89 box2~translateY=200
90 box2~rotate=45
91 box2~rotationAxis=point
92 box2~drawMode=drMo~line
93
94 -- add objects to the root node
95 root~getChildren~~add(box1)~~add(light)~~add(box2)
```

Listing 21: AmbientLight & DrawMode

At the beginning, the class "Box" is imported, access to "drawMode" is enabled and the class "Color" is loaded. Then a group is set as the root node.

Next, a box is created, its size and position are set and a rotation is added. Furthermore, this time the enum constant "FILL" is assigned to the box in line 68.

The next lines correspond to the example from chapter "4.9.1 AmbientLight".

In order to show the hardly visible edges of this box more clearly, a second box of the same size and in the same position as the first one is created. However, this box is assigned the enum constant "LINE". This makes the edges of the created box visible (see Figure 25).

The diagonal lines that can be seen in the output are the triangles that are drawn during the 3D rendering in JavaFX.

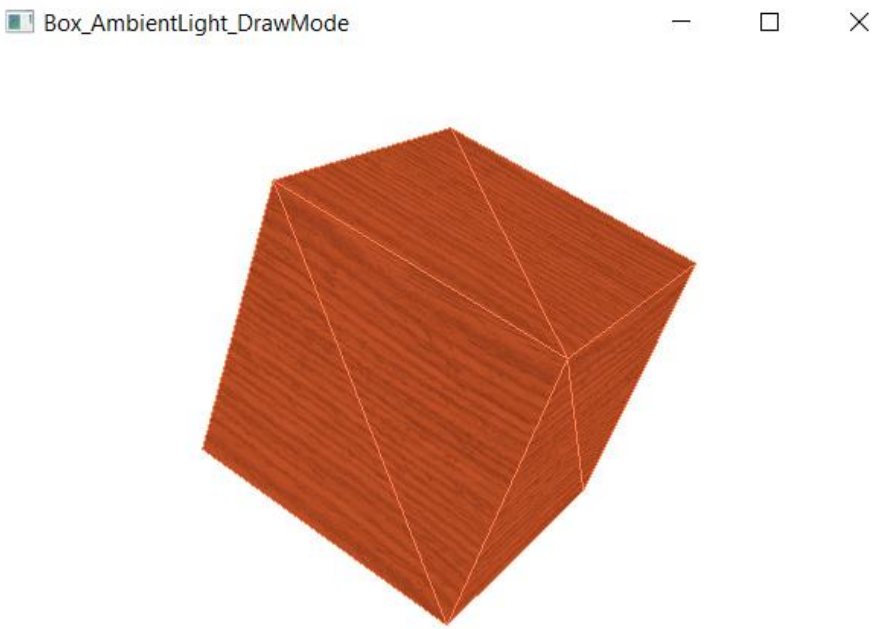

Figure 25: Output from AmbientLight & DrawMode

5 Conclusion

This bachelor thesis illustrated the programming of 3D objects in ooRexx exploiting JavaFX via BSF4ooRexx. The required software components were explained in more detail, installation instructions for them were given and then programmed nutshell examples were presented. With ooRexx and BSF4ooRexx and the bridge they build to Java, it is easier to use Java´s libraries and thus to program examples like the ones presented in the paper. In addition, it was shown how 3D shapes can be created and their properties, such as the material surface, can be defined and changed. There is the potential to create more complex programs in the future using JavaFX, ooRexx and BSF4ooRexx. For example, it would be possible to use MeshView, which can be used to create your own 3D shapes. Furthermore, adding animations to created objects would be conceivable. Since the user-friendliness of ooRexx is a clear advantage over programming with Java, the author believes it will certainly be used in the future.

References

Apache Software Foundation. (2004). *Apache License, Version 2.0*. Retrieved March 13, 2023, from https://www.apache.org/licenses/LICENSE-2.0

Autodesk. (n.d.). *Self-Illumination Map*. Retrieved April 16, 2023, from https://help.autodesk.com/view/3DSMAX/2024/ENU/?guid=GUID-0584ED4B-FE91-4B0B-A09C-22557D5D51DD

Flatscher, R. G. (2006). Resurrecting Rexx, Introducing Object Rexx.

Flatscher, R. G. (2012). Automatisierung mit ooRexx und BSF4ooRexx. *Proceedings der GMDS 2012 / Informatik 2012*, 1-12.

Flatscher, R. G. (2013). *Introduction to Rexx and ooRexx: From Rexx to Open Object Rexx (ooRexx) (1. ed.).* Facultas Verlags- und Buchhandels AG.

Flatscher, R. G. (2017). JavaFX for ooRexx - Creating Powerful Portable GUIs for ooRexx. *The Proceedings of the Rexx Symposium for Developers and Users*, 1-43.

Flatscher, R. G. (2023). *Creating Portable GUIs with JavaFX*. Retrieved March 13, 2023, from https://wi.wu-wien.ac.at/rgf/wu/lehre/autojava/material/foils/270_AutoJava_JavaFX_V04.pdf

Oracle. (n.d.-a). *Color (JavaFX 8)*. Retrieved March 14, 2023, from https://docs.oracle.com/javase/8/javafx/api/javafx/scene/paint/Color.html

Oracle. (n.d.-b). *CullFace (JavaFX 8)*. Retrieved March 24, 2023, from https://docs.oracle.com/javase/8/javafx/api/javafx/scene/shape/CullFace.html

Oracle. (n.d.-c). *DrawMode (JavaFX 8)*. Retrieved March 28, 2023, from https://docs.oracle.com/javase/8/javafx/api/javafx/scene/shape/DrawMode.html

Oracle. (n.d.-d). *Image (JavaFX 8)*. Retrieved April 12, 2023, from https://docs.oracle.com/javase/8/javafx/api/javafx/scene/image/Image.html

Oracle. (n.d.-e). *LightBase (JavaFX 8)*. Retrieved April 17, 2023, from https://docs.oracle.com/javase/8/javafx/api/javafx/scene/LightBase.html

Oracle. (n.d.-f). *Node (JavaFX 8)*. Retrieved March 20, 2023, from https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Node.html

Oracle. (n.d.-g). *PerspectiveCamera*. Retrieved March 15, 2023, from https://docs.oracle.com/javase/8/javafx/api/javafx/scene/PerspectiveCamera.html

Oracle. (n.d.-h). *PhongMaterial (JavaFX 8)*. Retrieved April 4, 2023, from https://docs.oracle.com/javase/8/javafx/api/javafx/scene/paint/PhongMaterial.html

Oracle. (n.d.-i). *Point3D (JavaFX 8)*. Retrieved March 22, 2023, from https://docs.oracle.com/javase/8/javafx/api/javafx/geometry/Point3D.html

Oracle. (n.d.-j). *Shape3D (JavaFX 8)*. Retrieved March 13, 2023, from https://docs.oracle.com/javase/8/javafx/api/javafx/scene/shape/Shape3D.html

Oracle. (n.d.-k). *Transform (JavaFX 8)*. Retrieved March 20, 2023, from https://docs.oracle.com/javase/8/javafx/api/javafx/scene/transform/Transform.html

Rexx Language Association. (n.d.). *About Open Object Rexx*. Retrieved April 21, 2023, from https://www.oorexx.org/about.html

Texturise. (2013-a). *Wood Fine Tiled: Normal*. Retrieved July 2, 2023, from http://2.bp.blogspot.com/-yId-zytBTeo/Ug6YqFgn8BI/AAAAAAAALA8/g_xWUApJPEw/s1600/wood-fine-NRM.jpg

Texturise. (2013-b). *Wood Fine Tiled: Spec*. Retrieved July 2, 2023, from http://2.bp.blogspot.com/-zBzWwdCFGuI/Ug6Ypq2xKII/AAAAAAAALA0/6wotxQ8cjQw/s1600/wood-fine-SPEC.jpg

Texturise. (2013-c). *Wood Fine Tiled: Wood*. Retrieved July 2, 2023, from http://2.bp.blogspot.com/-GfuzYF0lWNU/Ug6YrzFhtMI/AAAAAAAALBE/jtN_nPoMqEk/s1600/wood+fine.jpg

Texturise. (n.d.). *Licence*. Retrieved July 2, 2023, from http://www.texturise.club/p/licence_5.html

We Design Virtual. (2020). *What Does a Specular Map Do?* Retrieved April 16, 2023, from http://wedesignvirtual.com/what-does-a-specular-map-do/

Wikipedia. (2021-a). *Bumpmapping*. Retrieved April 16, 2023, from https://de.wikipedia.org/wiki/Bumpmapping

Wikipedia. (2021-b). *Culling*. Retrieved March 26, 2023, from https://de.wikipedia.org/wiki/Culling

Wikipedia. (2022). *JavaFX*. Retrieved February 17, 2023, from https://de.wikipedia.org/wiki/JavaFX

Wikipedia. (2023-a). *IntelliJ IDEA*. Retrieved April 28, 2023, from https://en.wikipedia.org/wiki/IntelliJ_IDEA

Wikipedia. (2023-b). *Java (programming language)*. Retrieved February 14, 2023, from https://en.wikipedia.org/wiki/Java_(programming_language)

Appendix

For the examples only partially presented in this paper, the full code is provided in the appendix.

A 1. Perspective Camera – Cylinder

Listing 22 shows the complete code for displaying the Cylinder with Perspective Camera from chapter 4.4.2.

```
37 -- setting title to the Stage
38 primaryStage~setTitle("Cylinder_PerspectiveCamera")
39
40 -- create the root node
41 root=.bsf~new("javafx.scene.Group")
42
43 -- create a cylinder
44 cylinder=.bsf~new("javafx.scene.shape.Cylinder")
45 -- set the radius and the height of the cylinder
46 cylinder~radius=50
47 cylinder~height=200
48 -- set position of the cylinder
49 cylinder~translateX=150
50 cylinder~translateY=150
51
52 -- add the object to the root node
53 root~getChildren~~add(cylinder)
54
55 -- create a perspective camera
56 camera=.bsf~new("javafx.scene.PerspectiveCamera")
57 -- set position of the camera
58 camera~translateX=0
59 camera~translateY=0
60 camera~translateZ=0
61
62 -- create a scene
63 scene=.bsf~new("javafx.scene.Scene", root, 800, 600)
64 -- add camera to the scene
65 scene~setCamera(camera)
66
```

```
67 -- put the scene on the stage
68 primaryStage~setScene(scene)
69 -- display the content of the stage
70 primaryStage~show
```

Listing 22: Perspective Camera – Cylinder

A 2. Perspective Camera – Sphere

Listing 23 shows the complete code for displaying the Sphere with Perspective Camera from chapter 4.4.2.

```
37 -- setting title to the Stage
38 primaryStage~setTitle("Sphere_PerspectiveCamera")
39
40 -- create the root node
41 root=.bsf~new("javafx.scene.Group")
42
43 -- create a sphere
44 sphere=.bsf~new("javafx.scene.shape.Sphere")
45 -- set the radius of the sphere
46 sphere~radius=75
47 -- set position of the sphere
48 sphere~translateX=150
49 sphere~translateY=150
50
51 -- add the object to the root node
52 root~getChildren~~add(sphere)
53
54 -- create a perspective camera
55 camera=.bsf~new("javafx.scene.PerspectiveCamera")
56 -- set position of the camera
57 camera~translateX=100
58 camera~translateY=100
59 camera~translateZ=-100
60
61 -- create a scene
62 scene=.bsf~new("javafx.scene.Scene", root, 800, 600)
63 -- add camera to the scene
64 scene~setCamera(camera)
65
```

```
66 -- put the scene on the stage
67 primaryStage~setScene(scene)
68 -- display the content of the stage
69 primaryStage~show
```

Listing 23: Perspective Camera - Sphere

A 3. Transformations – Scale

Listing 24 shows the complete code for adding a scale transformation from chapter 4.5.1.

```
37 -- setting title to the Stage
38 primaryStage~setTitle("Box_Scale")
39
40 -- create the root node
41 root=.bsf~new("javafx.scene.Group")
42
43 -- create a box
44 box=.bsf~new("javafx.scene.shape.Box")
45 -- set the width, height and depth of the box
46 box~width=150
47 box~height=150
48 box~depth=150
49 -- set position of the box
50 box~translateX=150
51 box~translateY=150
52 -- add a scale transformation to the box
53 box~scaleX=1.75
54 box~scaleY=0.75
55 box~scaleZ=0.5
56
57 -- add the object to the root node
58 root~getChildren~~add(box)
59
60 -- create a perspective camera
61 camera=.bsf~new("javafx.scene.PerspectiveCamera")
62 -- set position of the camera
63 camera~translateX=0
64 camera~translateY=0
65 camera~translateZ=0
66
```

```
67 -- create a scene
68 scene=.bsf~new("javafx.scene.Scene", root, 800, 600)
69 -- add camera to the scene
70 scene~setCamera(camera)
71
72 -- put the scene on the stage
73 primaryStage~setScene(scene)
74 -- display the content of the stage
75 primaryStage~show
```

Listing 24: Transformation - Scale

A 4. Transformations – Rotate

Listing 25 shows the complete for adding a rotation from chapter 4.5.2.

```
37 -- setting title to the Stage
38 primaryStage~setTitle("Box_Rotate")
39
40 -- create the root node
41 root=.bsf~new("javafx.scene.Group")
42
43 -- create a box
44 box=.bsf~new("javafx.scene.shape.Box")
45 -- set the width, height and depth of the box
46 box~width=150
47 box~height=150
48 box~depth=150
49 -- set position of the box
50 box~translateX=300
51 box~translateY=200
52 -- set rotation of the box
53 point=.bsf~new("javafx.geometry.Point3D", 1, 1, 1)
54 box~rotate=45
55 box~rotationAxis=point
56
57 -- add the object to the root node
58 root~getChildren~~add(box)
59
60 -- create a perspective camera
61 camera=.bsf~new("javafx.scene.PerspectiveCamera")
```

```
62 -- set position of the camera
63 camera~translateX=0
64 camera~translateY=0
65 camera~translateZ=0
66
67 -- create a scene
68 scene=.bsf~new("javafx.scene.Scene", root, 600, 400)
69 -- add camera to the scene
70 scene~setCamera(camera)
71
72 -- put the scene on the stage
73 primaryStage~setScene(scene)
74 -- display the content of the stage
75 primaryStage~show
```

Listing 25: Transformation - Rotate

A 5. PhongMaterial – BumpMap

Listing 26 shows the complete code for creating a box with added bump map from chapter 4.8.3.

```
37 -- change directory to program location
38 parse source . . s
39 call directory filespec('loc', s)
40
41 -- setting title to the Stage
42 primaryStage~setTitle("Box_BumpMap")
43
44 -- create the root node
45 root=.bsf~new("javafx.scene.Group")
46
47 -- create a box
48 box=.bsf~new("javafx.scene.shape.Box")
49 -- set the width, height and depth of the box
50 box~width=150
51 box~height=150
52 box~depth=150
53 -- set position of the box
54 box~translateX=300
55 box~translateY=200
```

```
56 -- set rotation of the box
57 point=.bsf~new("javafx.geometry.Point3D", 1, 1, 1)
58 box~rotate=45
59 box~rotationAxis=point
60
61 -- load an image
62 img=.bsf~new("javafx.scene.image.Image", "file:normal.jpg")
63
64 -- preparing the material
65 material=.bsf~new("javafx.scene.paint.PhongMaterial")
66 material~bumpMap=img
67
68 -- set the material to box
69 box~setMaterial(material)
70
71 -- add the object to the root node
72 root~getChildren~~add(box)
73
74 -- create a perspective camera
75 camera=.bsf~new("javafx.scene.PerspectiveCamera")
76 -- set position of the camera
77 camera~translateX=0
78 camera~translateY=0
79 camera~translateZ=0
80
81 -- create a scene
82 scene=.bsf~new("javafx.scene.Scene", root, 600, 400)
83 -- add camera to the scene
84 scene~setCamera(camera)
85
86 -- put the scene on the stage
87 primaryStage~setScene(scene)
88 -- display the content of the stage
89 primaryStage~show
```

Listing 26: PhongMaterial - BumpMap

A 6. PhongMaterial – SpecularMap

Listing 27 shows the complete code for creating a box with added specular map from chapter 4.8.4.

```
-- change directory to program location
parse source . . s
call directory filespec('loc', s)

-- setting title to the Stage
primaryStage~setTitle("Box_SpecularMap")

-- create the root node
root=.bsf~new("javafx.scene.Group")

-- create a box
box=.bsf~new("javafx.scene.shape.Box")
-- set the width, height and depth of the box
box~width=150
box~height=150
box~depth=150
-- set position of the box
box~translateX=300
box~translateY=200
-- set rotation of the box
point=.bsf~new("javafx.geometry.Point3D", 1, 1, 1)
box~rotate=45
box~rotationAxis=point

-- load an image
img=.bsf~new("javafx.scene.image.Image", "file:spec.jpg")

-- preparing the material
material=.bsf~new("javafx.scene.paint.PhongMaterial")
material~specularMap=img

-- set the material to box
box~setMaterial(material)

-- add the object to the root node
root~getChildren~~add(box)

```

```
74 -- create a perspective camera
75 camera=.bsf~new("javafx.scene.PerspectiveCamera")
76 -- set position of the camera
77 camera~translateX=0
78 camera~translateY=0
79 camera~translateZ=0
80
81 -- create a scene
82 scene=.bsf~new("javafx.scene.Scene", root, 600, 400)
83 -- add camera to the scene
84 scene~setCamera(camera)
85
86 -- put the scene on the stage
87 primaryStage~setScene(scene)
88 -- display the content of the stage
89 primaryStage~show
```

Listing 27: PhongMaterial - SpecularMap

A 7. PhongMaterial – SelfIlluminationMap

Listing 28 shows the complete code for creating a box with added self-illumination map from chapter 4.8.5.

```
37 -- change directory to program location
38 parse source . . s
39 call directory filespec('loc', s)
40
41 -- setting title to the Stage
42 primaryStage~setTitle("Box_SelfIlluminationMap")
43
44 -- create the root node
45 root=.bsf~new("javafx.scene.Group")
46
47 -- create a box
48 box=.bsf~new("javafx.scene.shape.Box")
49 -- set the width, height and depth of the box
50 box~width=150
51 box~height=150
52 box~depth=150
53 -- set position of the box
```

```
54 box~translateX=300
55 box~translateY=200
56 -- set rotation of the box
57 point=.bsf~new("javafx.geometry.Point3D", 1, 1, 1)
58 box~rotate=45
59 box~rotationAxis=point
60
61 -- load an image
62 img=.bsf~new("javafx.scene.image.Image", "file:wood.jpg")
63
64 -- preparing the material
65 material=.bsf~new("javafx.scene.paint.PhongMaterial")
66 material~selfIlluminationMap=img
67
68 -- set the material to box
69 box~setMaterial(material)
70
71 -- add the object to the root node
72 root~getChildren~~add(box)
73
74 -- create a perspective camera
75 camera=.bsf~new("javafx.scene.PerspectiveCamera")
76 -- set position of the camera
77 camera~translateX=0
78 camera~translateY=0
79 camera~translateZ=0
80
81 -- create a scene
82 scene=.bsf~new("javafx.scene.Scene", root, 600, 400)
83 -- add camera to the scene
84 scene~setCamera(camera)
85
86 -- put the scene on the stage
87 primaryStage~setScene(scene)
88 -- display the content of the stage
89 primaryStage~show
```

Listing 28: PhongMaterial - SelfIlluminationMap

A 8. PhongMaterial – Combination

Listing 29 shows the complete code for the combination of diffuseMap, bumpMap, specularMap and specularPower from chapter 4.8.6.

```
37 -- change directory to program location
38 parse source . . s
39 call directory filespec('loc', s)
40
41 -- setting title to the Stage
42 primaryStage~setTitle("Box_PhongMaterial")
43
44 -- create the root node
45 root=.bsf~new("javafx.scene.Group")
46
47 -- create a box
48 box=.bsf~new("javafx.scene.shape.Box")
49 -- set the width, height and depth of the box
50 box~width=200
51 box~height=200
52 box~depth=200
53 -- set position of the box
54 box~translateX=300
55 box~translateY=200
56 -- set rotation of the box
57 point=.bsf~new("javafx.geometry.Point3D", 1, 1, 1)
58 box~rotate=90
59 box~rotationAxis=point
60
61 -- load images
62 img=.bsf~new("javafx.scene.image.Image", "file:wood.jpg")
63 img2=.bsf~new("javafx.scene.image.Image", "file:normal.jpg")
64 img3=.bsf~new("javafx.scene.image.Image", "file:spec.jpg")
65
66 -- preparing the material
67 material=.bsf~new("javafx.scene.paint.PhongMaterial")
68 material~diffuseMap=img
69 material~bumpMap=img2
70 material~specularMap=img3
71 material~specularPower=15
72
73 -- set the material to box
```

```
box~setMaterial(material)

-- add the object to the root node
root~getChildren~~add(box)

-- create a perspective camera
camera=.bsf~new("javafx.scene.PerspectiveCamera")
-- set position of the camera
camera~translateX=0
camera~translateY=0
camera~translateZ=0

-- create a scene
scene=.bsf~new("javafx.scene.Scene", root, 600, 400)
-- add camera to the scene
scene~setCamera(camera)

-- put the scene on the stage
primaryStage~setScene(scene)
-- display the content of the stage
primaryStage~show
```

Listing 29: PhongMaterial – Combination

A 9. LightBase – PointLight

Listing 30 shows the complete code for creating a point light from chapter 4.9.2.

```
-- change directory to program location
parse source . . s
call directory filespec('loc', s)

-- setting title to the Stage
primaryStage~setTitle("Box_PointLight")

-- get access to the JavaFX colors
col=bsf.loadClass("javafx.scene.paint.Color")

-- create the root node
root=.bsf~new("javafx.scene.Group")

```

```
50 -- create a box
51 box=.bsf~new("javafx.scene.shape.Box")
52 -- set the width, height and depth of the box
53 box~width=200
54 box~height=200
55 box~depth=200
56 -- set position of the box
57 box~translateX=300
58 box~translateY=200
59 -- set rotation of the box
60 point=.bsf~new("javafx.geometry.Point3D", 1, 1, 1)
61 box~rotate=45
62 box~rotationAxis=point
63
64 -- load an image
65 img=.bsf~new("javafx.scene.image.Image", "file:wood.jpg")
66
67 -- preparing the material
68 material=.bsf~new("javafx.scene.paint.PhongMaterial")
69 material~diffuseMap=img
70
71 -- set the material to box
72 box~setMaterial(material)
73
74 -- create a point light
75 light=.bsf~new("javafx.scene.PointLight")
76 light~color=col~WHITE
77 -- set position of the light
78 light~translateX=500
79 light~translateY=100
80 light~translateZ=-100
81
82 -- add objects to the root node
83 root~getChildren~~add(box)~~add(light)
84
85 -- create a perspective camera
86 camera=.bsf~new("javafx.scene.PerspectiveCamera")
87 -- set position of the camera
88 camera~translateX=0
89 camera~translateY=0
90 camera~translateZ=0
```

```
 91
 92 -- create a scene
 93 scene=.bsf~new("javafx.scene.Scene", root, 600, 400)
 94 -- add camera to the scene
 95 scene~setCamera(camera)
 96
 97 -- put the scene on the stage
 98 primaryStage~setScene(scene)
 99 -- display the content of the stage
100 primaryStage~show
```

Listing 30: LightBase - PointLight

A 10. LightBase – AmbientLight & DrawMode

Listing 31 shows the complete code for the combination of AmbientLight and drawMode from chapter 4.9.3.

```
 37 -- change directory to program location
 38 parse source . . s
 39 call directory filespec('loc', s)
 40
 41 -- setting title to the Stage
 42 primaryStage~setTitle("Box_AmbientLight_DrawMode")
 43
 44 -- import class Box
 45 box=bsf.import("javafx.scene.shape.Box")
 46 -- get access to drawMode
 47 drMo=bsf.loadClass("javafx.scene.shape.DrawMode")
 48 -- get access to the JavaFX colors
 49 col=bsf.loadClass("javafx.scene.paint.Color")
 50
 51 -- create the root node
 52 root=.bsf~new("javafx.scene.Group")
 53
 54 -- create a box
 55 box1=box~new
 56 -- set the width, height and depth of the box
 57 box1~width=200
 58 box1~height=200
 59 box1~depth=200
```

```
60  -- set position of the box
61  box1~translateX=300
62  box1~translateY=200
63  -- set rotation of the box
64  point=.bsf~new("javafx.geometry.Point3D", 1, 1, 1)
65  box1~rotate=45
66  box1~rotationAxis=point
67  -- set the value of drawMode
68  box1~drawMode=drMo~fill
69
70  -- load an image
71  img=.bsf~new("javafx.scene.image.Image", "file:wood.jpg")
72
73  -- preparing the material
74  material=.bsf~new("javafx.scene.paint.PhongMaterial")
75  material~diffuseMap=img
76
77  -- set the material to box
78  box1~setMaterial(material)
79
80  -- create an ambient light
81  light=.bsf~new("javafx.scene.AmbientLight")
82  light~color=col~CORAL
83
84  box2=box~new
85  box2~width=200
86  box2~height=200
87  box2~depth=200
88  box2~translateX=300
89  box2~translateY=200
90  box2~rotate=45
91  box2~rotationAxis=point
92  box2~drawMode=drMo~line
93
94  -- add objects to the root node
95  root~getChildren~~add(box1)~~add(light)~~add(box2)
96
97  -- create a perspective camera
98  camera=.bsf~new("javafx.scene.PerspectiveCamera")
99  -- set position of the camera
100 camera~translateX=0
```

```
101 camera~translateY=0
102 camera~translateZ=0
103
104 -- create a scene
105 scene=.bsf~new("javafx.scene.Scene", root, 600, 400)
106 -- add camera to the scene
107 scene~setCamera(camera)
108
109 -- put the scene on the stage
110 primaryStage~setScene(scene)
111 -- display the content of the stage
112 primaryStage~show
```

Listing 31: LightBase - AmbientLight & DrawMode