Politechnika Wrocławska

Faculty of Information and Communication Technology

Field of study: INFAN

Bachelor Thesis

# MODULE FOR THE VISUALIZATION AND ASSEMBLY OF VIDEO SEQUENCES FOR EXPERIMENTING WITH DRIVING ALGORITHMS OF AUTONOMOUS VEHICLES

Vladyslav Gavryliuk

keywords:
Computer Graphics,
Video Rendering,
3D visualization

short summary:
Thesis describes the implementation of a visualization 3D world for experimenting with driving algorithms of autonomous vehicles. The solution provides a choice for user, such as select the renderer for specific area of the world. The module developed by the author aid the users to experiment with vehicles driving algorithm without worrying a real-world damage.

| | |
|---|---|
| Supervisor | Dr. Jerzy Sas, PhD. |
| | Title/ degree/ name and surname |

*For the purposes of archival thesis qualified to: **

a) *Category A (perpetual files)*
b) *Category BE 50 (subject to expertise after 50 years)*

** Delete as appropriate*

stamp of the faculty

Wrocław, 2022

## TABLE OF CONTENTS

## ABSTRACT

Nowadays 3D technologies evolve in geometric progression, anything that people are able to imagine may be viewed where they want, beginning from websites and finishing with the simulator of virtual reality.

The subject is to create a high-performance application that provides the possibility to store the obtained view of the scene based on an autonomous vehicle driving algorithm as a single image or assembly of video sequences. The environment and related technologies would be selected as much as possible optimal for implementing the visualization simulator. Along with it, the obj file format would be extended to store additional information to increase efficiency.

By the end of the whole work the selected technology will be discussed, the problems that came across during development and testing over the application.

# INTRODUCTION

Autonomous vehicles are populating the world, as one of the most known companies is Tesla. Thereby, the demand for them is increasing. Therefore, the algorithms and their evaluation should be trained and tested as precisely as it is possible.

The camera projection should be placed on the car to observe the behavior around the specific vehicle and concerning the information decide the specific way. Thus, to provide a car and train the algorithm Moreover, experimenting with those algorithms in the real world may provide to crushes and planet damage. Therefore, it would be a good point to first create a virtual environment where tests related to the specific rules would be applied. At that time, the camera could be assigned. In order to verify the algorithm with different regulations, the video might be retrieved from each simulation. Therefore, the view of the rendering process should be defined reasonably precisely and as much as realistic with the technique of computer graphics. Meanwhile, for better usage, it should be simulated in real-time.

This is a complex problem and is divided into several modules, where one of the most important is a module of visualization.

## THESIS GOAL

The aim of the project described here is to design and implement a rendering module for a traffic modeling and visualization system. There are many 3D renderers available as commercial products or open-source software. Although there are many 3D renderers available as non-proprietary software or commercial products, they do not seem to be well suited to the specific requirements of a traffic modeling system. Elements that are not quite solved in other available software include the following: reuse of scene data in subsequent rendering frames, such as divide the objects into static and dynamic ones, dynamic control of affine transformations that determine the positions of moving objects whose internal geometry does not change from frame to frame, simple formatting of scene data accepted by the renderer so that the scene description can be easily created by other modules responsible for building the description based on GIS data. Thus, the product resulting from the described project is not just a pure single static frame renderer, but a program that controls the rendering process of the entire animation sequence, where the implemented single frame renderer is just a kind of plug-in that can be used for rendering a single frame in the animation.

## THESIS SCOPE

The first chapter describes the overview of the whole system, the analysis of the already existing solution. There is also included the comparison of relative environments and explanation of the author's selection. Provided information regarding the standards, theoretical background, external components, and summary of those topics.

The further chapter contains design elements, such as listed required functionalities, specification of use cases, the architecture of independent author's module with the description of the implementation, a couple of diagrams which presents the interacts with specific sections, the data structure used inside and outside the project, UI design.

The fourth chapter describes the installation process and user manual. How to execute the application from the user side and which abilities this program supports. There is included navigation through each UI component.

The fifth one describes the testing of the system. It provides information about which tests were involved and what performance it does.

The last chapter is about the conclusion and future works. There are topics about summary up of achieved results and future recommendations that should be taken into account when someone would be developing new features.

# 1. GLOSSARY

**Affine transformation** - a linear mapping method that preserves points, straight lines, and planes. Sets of parallel lines remain parallel after an affine transformation. There are 4 types of transformation, but using 3 of them are fully enough to satisfy our requirements, such as translatation, rotation and scale.

**Blending** - the purpose of this to implement transparency within objects. Transparency have a combination of colors from the mesh itself and other mesh behind it with varying intensity.

**Batch** script - simple text file that contains certain commands that are executed in order.

**Camera** - A virtual camera from which rendering is performed.

**Computer graphics** - a field of computer science that studies methods of digitally synthesising and manipulating visual content.

**Culling test** - skips the primitives that are away from the viewer.

**Depth test** - stores all the fragment colors as the visual output, stores information per fragment and has the same width and height as the color buffer.

**Frame** - are rectangular areas meant for inserting graphics and text. They allow users to place objects wherever they want to on the page. In video and animation, frames are individual pictures in a sequence of images.

**Geometry** - typically used to refer to vertex  rendering primitive connectivity information.

**Homogeneous coordinates** - a system of coordinates used in projective geometry, as Cartesian coordinates are used in Euclidean geometry.

**Lighting** - computations simulating the behavior of light.

**Mesh** - is a 3D object representation consisting of a collection of vertices and polygons.

**Mtl** file - a material settings file.

**Normalized device coordinate** - shortly NDC space that defines a screen independent display coordinate system.

**Obj** file - a standard 3D file format.

**Polygon** - any closed curve consisting of a set of line segments (sides) connected such that no two segments cross.

**Pixel** - smallest element of a raster image.

**Primitive** - a basic unit of geometry for rendering or modelling.

**Raster graphics** - graphics represented as a rectangular grid of pixels.

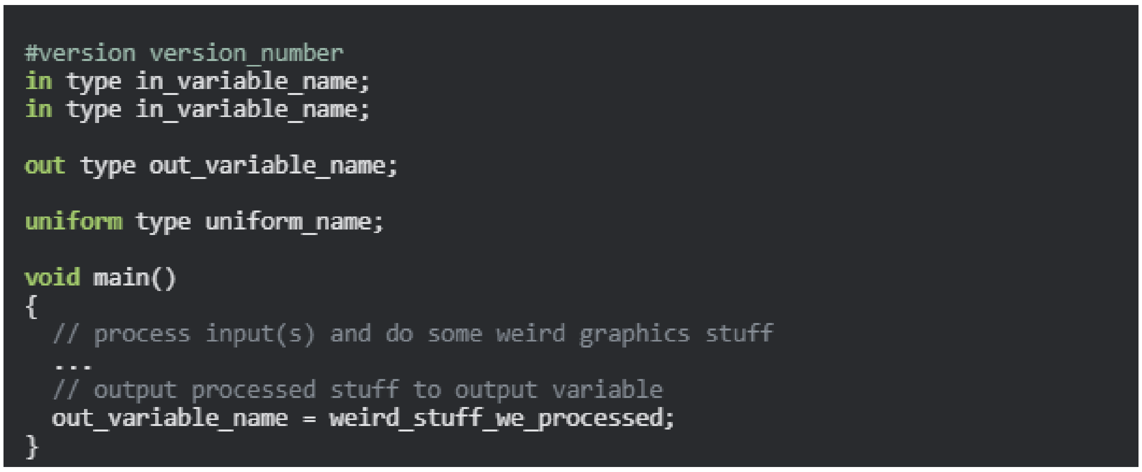
**Rasterization** - Converting vector graphics to raster graphics.

**Reflection** - the value of this in 3D graphics emulates reflective objects such as mirrors and shiny surfaces.

**Render** - is a particular view of a 3D model that has been converted from any coded content into a realistic image and includes some lighting and sophisticated effects.

**Scene** - is a collection of objects, where each of them compose of its own characteristics.

**Shaders** - are written in the C-like language GLSL. **GLSL** is tailored for use with graphics and contains useful features specifically targeted at vector and matrix manipulation. Shaders always begin with a version declaration, followed by a list of input and output variables, uniforms and its main function. Each shader's entry point is at its main function where we process any input variables and output the results in its output variables.Vertex shader used to manipulate with data and return the value of color, so then fragment shader takes that value and adds it to the scene corresponding the specific mesh or part of the mesh. A shader typically has the following structure as shown below on 1.1



```
#version version_number
in type in_variable_name;
in type in_variable_name;

out type out_variable_name;

uniform type uniform_name;

void main()
{
  // process input(s) and do some weird graphics stuff
  ...
  // output processed stuff to output variable
  out_variable_name = weird_stuff_we_processed;
}
```

Fig. 1.1. Shader example [21]

**Triangle primitive** - The most common rendering primitive defining triangle meshes, rendered by graphics processing units.

**Vector** - a mathematical concept. Crudely, it represents a set of geometrical primitives and a displacement in some coordinate space.

**Vertex** - an element of some 3D geometry which typically has a position and some other attributes (color, positions, textures, normals).The position of a vertex (and thus sometimes the vertex itself) can be represented by a vector if one assumes the vector is a displacement from the origin of the coordinate system.

**Window** - a rectangular region of a screen.

# 2. OVERVIEW OF EXISTING SOLUTIONS

## 2.1. OVERVIEW OF THE WHOLE SIMULATION SYSTEM

This chapter involved only describes more precisely how the whole system work. It was mentioned earlier that due to the complexity of the project, the whole system is divided into four independent modules. In view of the fact that the system communication of all the modules the exchange between them is in the files, as we have come up with. Thus, to achieve it as the system, the simple interface was managed with four buttons, by clicking the specific one, it launches the related module. Moreover, the modules are developed in different environments, as shown in the figure 2.1, nevertheless, the technologies that were chosen by the team members are not decisive in the communication. Therefore, to avoid unforeseen problems, an executable (.exe) file has been generated for each module. In order to, the user should not care about installing extraneous components.

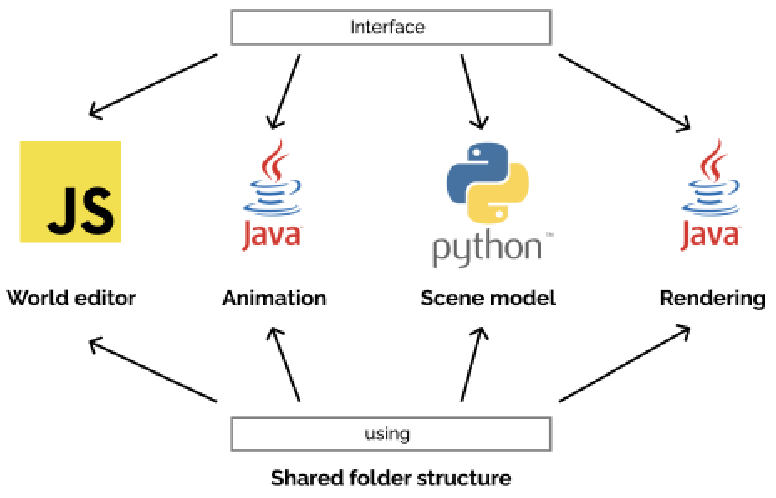


Fig. 2.1. Interaction between modules

Thus, the author receives the files that contain data that were processed through 3 modules. The whole interaction is divided into 4 modules, so to clarify *what does* each module the following table 2.1 created.

| Module name | input | output | Responsibility |
|---|---|---|---|
| World editor | GIS data | geojson | description of the static world, with some manipulations of the objects presented on the map. |
| Animation scenario | geojson | geojson | defining the rules and generation the animation, make the scenarios. |
| Generator | geojson | extended obj and mtl | generating the files with corresponding matrices of transformation. |
| Rendering | extended obj and mtl | displayed scene | handling the received files and visualizing the final 3D world simulation. |

Tab. 2.1. Table of presenting the work of each module.

The first module is the communication between world editor and second one called animation. In the world editor stage the are of map is selected and then additional manipulations may be proceed. For example, the adding or deleting of specific objects (roads, building, traffic lights, etc) and then it goes through animation part which produce the vehicles behaviours based on the information of the received map. This interacts shown in figure 2.2.

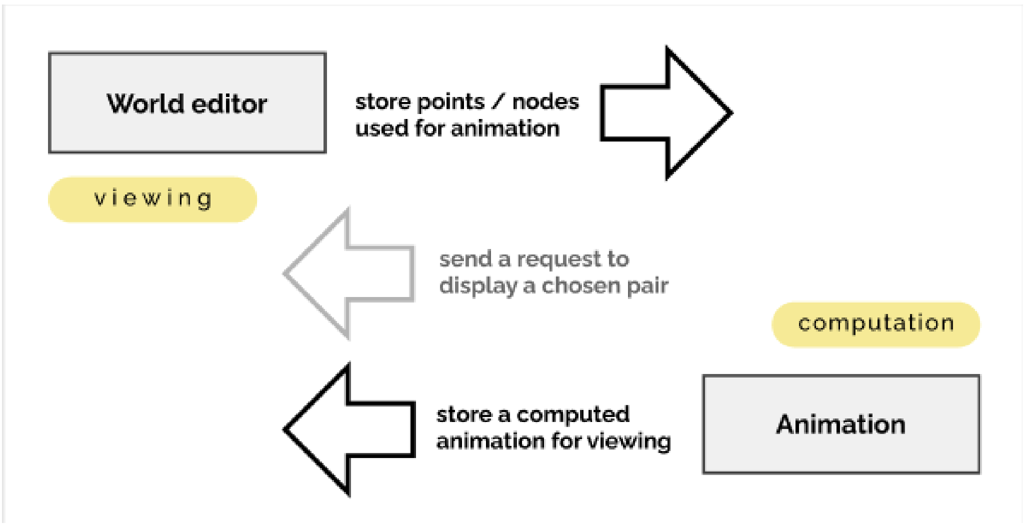

World editor
store points / nodes used for animation
viewing
send a request to display a chosen pair
computation
store a computed animation for viewing
Animation

Fig. 2.2. 1st stage

The third module is the interaction between a generator of 3D world and fourth one called visualizer based on received files. Those files are converted into frames and displayed on the window. The generator support opportunity to add instances in places they may relate to, for example, trees are corresponds to parks, jungles, forests, etc. Those information is written in extended obj files. Therefore, in the end render provides nicely view of frame by frame. This communication described in figure 2.3.

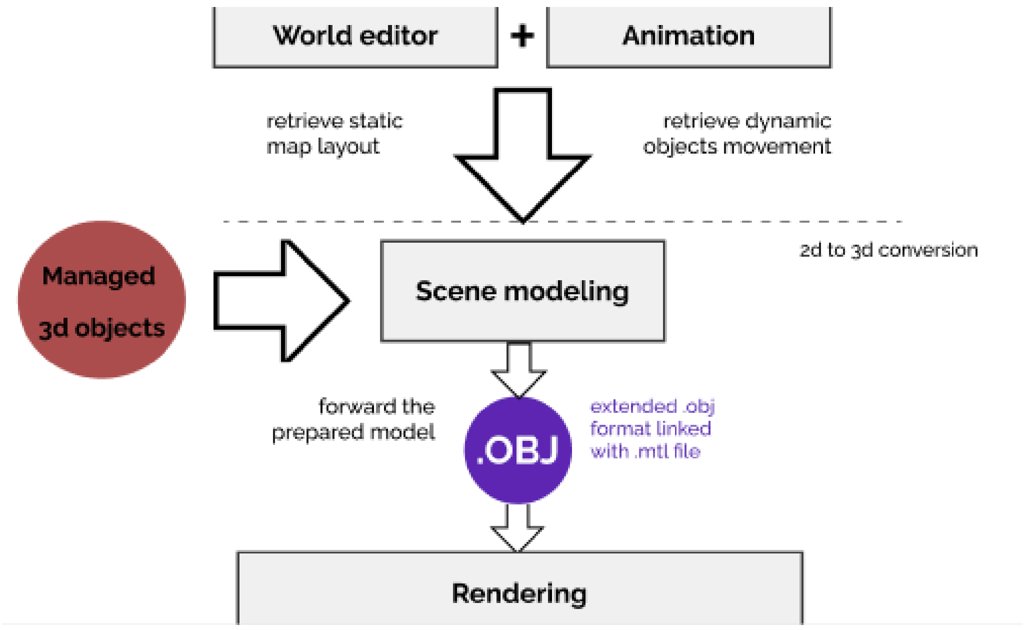

World editor
+
Animation
retrieve static map layout
retrieve dynamic objects movement
2d to 3d conversion
Managed 3d objects
Scene modeling
forward the prepared model
.OBJ
extended .obj format linked with .mtl file
Rendering

Fig. 2.3. 2nd stage

In order to clarify the impact of each module, the following explanations are considered.

On the first level, there are "playgrounds" for various animations shown in figure 2.4. The playground is the description of the static part of the virtual world containing terrain, streets, buildings, and their attributes. They are created using the module responsible for

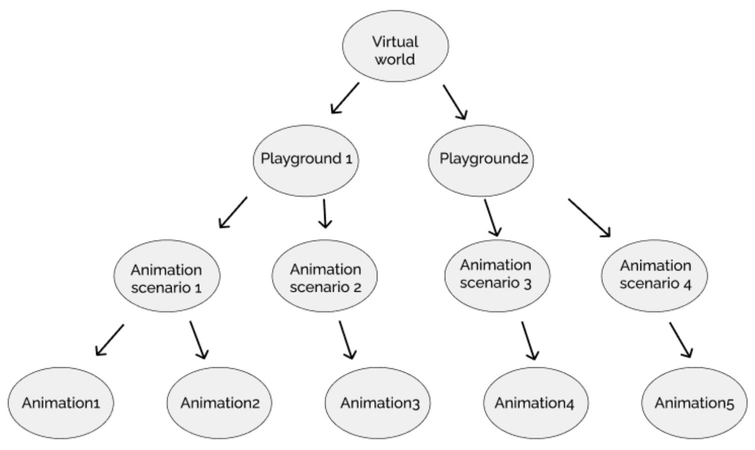


Fig. 2.4. The graph of the whole system

static world model creation. The "playground data" consist of the dataset consistent with GIS formats and its counterpart relevant to be the input to the visualization module. IN this project it is assumed to use the extended obj file format to store the static world model.

On the next level "animation scenarios" are stored. By animation scenario we mean the data set describing the principles av animated traffic in the playground. The animation scenario defines the source and destination points in the static world where the moving vehicles are inserted into the world and to which they travel. It also defines the stochastic properties of sources [16], so that it is possible to control the traffic intensity. For each playground and scenario, a number of animations can be created. The individual animation is the sequence of world states in subsequent animation frames. The world state in a specific frame defines the momentary position of all traveling cars in the specific time moment of the animation time interval. Because animation is a random process the sequences of world states in various animations created from the same animation scenarios can be different.

The animation as the sequence of world states stored in the specific file is the start point for the creation of the complete scene description which is used to render the specific image frame. The animation control module calls the GIS->OBJ creation module which creates data for the specific frame which next is passed to the rendering module.

In summary, the whole system provides end-to-end integration. It goes through each module that provides some modifications on the map, as a result, the author's module would simulate the visualization of some animation on the predefined area of the real world. The author is responsible only for the last stage called Rendering and the further chapters with descriptions related to that module would be involved.

## 2.2. GENERAL REVIEW OF THE SIMILAR PRODUCTS

There are various similar solutions, at least hundreds or even thousands can be counted as existed visualizers of 3D scene, some faster, some slower.

In order to compare with already existing products, there were selected three examples that are more relevant to the problem faced by the author. Those are Pov-Ray, AirSim and Lumion.

### 2.2.1. Pov-Ray

Persistence of Vision Ray Tracer is a high-quality ray tracing program for creating the 3D graphics that views very realistic [9]. The first release was around 30 years ago and still updates, despite of stable version 3.7.0 released in 2013. The program is written in C++ and accepts its internal data format with extension .pov. Pov-Ray based on drawing pixel by pixel that leads to low efficiency by comparing with another projects.

The strong sides :

- provides the library of ready-made objects, textures and scenes.
- maintain geometric primitives.
- different types of lighting sources.
- support real atmospheric effects, for instance clouds.
- rendered output may be stored in different types.

### 2.2.2. AirSim

The software created by Microsoft and responsible for simulation of different vehicles, built by Unreal Engine with APIs from another source codes written in C++, C sharp, C, Python [1]. The initial release is the February 16, 2017 and has stable updating, last stable release was less than 4 months ago. The idea of the program is to experiment with deep learning.

The strong sides :

- provides possibility for testing of autonomous solutions without worrying about real-world damage.
- supports deep learning.
- may include several cameras, for example at the same monitoring front and back of the car.
- view can be displayed in different mode, such as colorful, white-black, midnight, etc.

### 2.2.3. Lumion

Lumion is based on the principle of reducing the effort required on end to transform static, 3D designs into 360 panoramas, images, and videos [5]. This technology is compatible with all of the major CAD and 3D modeling programs. This means that, for instance, the whole scene might be imported in several parts, as a residential home design from AutoCad and a landscape design from 3DS MAX into the same Lumion project. The initial release in December 2010 with stable up to day, nevertheless the last stable version was published in November 2019. After model imported, it proposes user-friendly tools to help setting the design.

The strong sides :

- wide compatibility.
- real-time rendering with Lumion and several 3D modeling programs.
- intuitive project editing tools
- Cinematic animations.

#### 2.2.4. Comparison with Author's render

Considering the information described each application and its strongest advantages, the following table 2.2 created. This table concerns of most crucial features provided by solutions. The OS that supports for author's renderer and Lumion is Windows, meanwhile, AirSim and Pov-Ray are supporting Windows, Linux, and macOS.

| Feature of the program | Pov-Ray | AirSim | Lumion |
|---|---|---|---|
| Open-source | yes | no | no |
| Stable updating | no | yes | no |
| Free to use | yes | yes | no |
| Efficiency | low | high | high |
| define the geometry as static and render | no | no | no |
| Dynamic management of affine transformations | no | no | no |
| Accepting of external file formats | yes | no | no |
| Reuse of scene data | no | no | no |
| Plug-in ability | yes | no | yes |
| Vehicle autonomous driving algorithm | no | yes | yes |
| Dynamic camera | no | yes | yes |

Tab. 2.2. Table of comparing programming languages.

Summarizing everything, the solution proposed by the author would have some unique features or be more adaptive than those already presented. Moreover, the scene geometry in existing solutions does not divide into static and dynamic, which does not allow the possibility of the reuse of scene data in subsequent frame rendering. There are also missing the dynamic management of affine transformations defining positions of moving objects which the same geometry from frame to frame, simplicity of the scene data formats accepted by the renderer, so that the scene description can be easily created by other modules responsible for scene description building from GIS data. Thus, the difference in some features is significant and leads to justifies a rewrite.

## 2.3. COMPARING PROGRAMMING LANGUAGES FOR IMPLEMENTATION THE APPLICATION

Modern programming allows many approaches to solving any task, resulting in a large set of comparisons of advantages and disadvantages. Thus, the task at hand can be realized using many technologies, but the idea is to choose the most appropriate one.

### 2.3.1. C

The construction of C is not typical for today most popular languages, as it does not support such features as polymorphism, encapsulation, and inheritance that leads to that this programming language is not OOP (object oriented programming), in view of fact that C is related to procedural one. Therefore, the data is not hidden, as the reason it focuses on methods or processes. Moreover, C is known as a subset C++ which will be described in next section.

However, it does not mean that C as language is no relevant anymore for today tasks, especially for developing the computer graphics with utilizing the native library OpenGL [8]. C language still has inquiry in producing 3D graphics. Obvious, that demand is not so high at it was even 10-30 years ago, since other languages popularize and provides additional feature which improves the performance and easy-understanding the concept of projects.

### 2.3.2. C++

The C++ programming language was mentioned before and famous as superset of C, there were provided many new features, particularly classes, that is why C++ has another name as "C with Classes". The purpose of creating this technology was to increase dimension of possible resolved solutions. Thus, C++ is regular used for solving modern problems with guarantee high performance, though another innovated languages slowly crowd out this technology of the market.

Consequently, due to its existing were created various thousands of projects related to sphere as computer graphics, including 3D renders which were produced in different render techniques, for instance ray tracing, ray casting, z-buffer and so on and so fourth.

### 2.3.3. Java

Java language of programming is gaining momentum, as high-level, class-based,object-oriented programming language that is designed to have as few implementation dependencies as possible and corresponds to java enterprise edition that supplies for large-scable applications, such as distributed computing and web services. The syntax is close to the C and C++, nevertheless stated as fewer lower-level language. The guge breakthrough has come with Java 8 release, there were adjusted following features : supports for functional programming, new APIs for date time manipulation, new streaming API, etc.

The necessity as Java programming language to solve the problem is increasing day per day, thankful its flexibility and wide range of functionality. Accordingly, leads to a high demand of developers in the IT market. LWJGL API is the most popular for Java in context of 3D graphics, as it provides many features that gives for developers more flexibility. [6].

### 2.3.4. Python

Python is known as the scripting programming language, despite the fact that it is classified as OOP, there are a lot of weaknesses, for example the fewer data types, not memory efficient, runtime errors, slow speed as it is interpreted and dynamically-typed language.

PyOpenGL API was invoked to use OpenGL APIs in Python environment, notwithstanding that approach is not enough for covering all CPU-side code [10].

#### 2.3.5. JavaScript

Javasript has abbreviature JS , is a programming language that is one of core languages for Web development. Js is a high-level with dynamic typing, prototype-based object-orientation and first-class functions. Moreover, originally it was created for creating server-side, in spite of this today exist more components and some of them might produce variety of application for servers. The most relevant and popular for this purpose is Node.js.

Javascript uses another way to create 3D graphics, more known as WebGL. By comapring WebGL [13] and OpenGL, there are significant difference, as the result first technology provides less functionalities and efficiency concerning second one.

#### 2.3.6. Selection of the technology for the realization the application

Having summarised all of the above, a table of the best technologies that can solve the problem has been compiled for a better overview.

The languages comparison shown in table 2.3. OpenGL support - simply means that we may easily interacts with OpenGL methods, functions, classes, interfaces, etc.

| | C | C++ | Java | Python | JS |
|---|---|---|---|---|---|
| Opengl support | yes | yes | yes | yes | yes |
| Libraries for obj reader | no | no | yes | yes | no |
| Efficiency | high | high | high | medium | medium |
| Ease of learning | hard | medium | medium | easy | easy |
| API GUI | no | no | yes | yes | yes |

Tab. 2.3. Table of comparing programming languages.

Shortly, Java provides all features that would be needed for all requirements and satisfies the principle of high-performance application. Moreover, demand in IT market with Java position during last years located in top three. Therefore, the chance of finding further developers is greater.

### 2.4. THEORETICAL BACKGROUND

There are various visualization techniques of computer graphics, such as scan line, ray casting, ray tracing, etc. Some of them are outdated or draw scenes according to old principles, such as very simple geometry with a basic color without supporting any additional effects. Therefore a scan line was chosen as a suitable approach, as it corresponds to the rasterization of subsequent triangles and pixel filling after rasterization and is relevant to modeling applications. The scanning line provides a large set of possibilities to obtain a realistic view, but at the same time, it reduces efficiency. Thus, we have to find a compromise between the final view, runtime, and ease of implementation of such a renderer. Moreover, OpenGL implements the scanning line technique, so this approach was chosen without any doubt.

As the render based on OpenGL is a good point to clarify that it contains of two different paradigms, the oldest versions up to 1.5 were using fixed functions pipeline, so from the version 2.0 the maintenance had modified for more flexibility of users and named as PFP (programmed functions pipeline), as the reason the PFP principle selected for current solution.

Thus, we have various approaches for further implementation. There is a huge set of lighting models for 3D image rendering. With the object of creation nicely scene was selected one of the most popular lighting models named by B.T. Phong([1]). Despite its simplicity, the Phong model is gifted to render quite realistic images about simple surfaces.

The Phong model reproduces the most typical optical phenomena:

a) self-luminosity of the surface.

b) diffuse reflection of ambient light.

c) diffuse reflection on Lambertian surfaces.

d) specular reflection of primary light on glossy surfaces.

For the purpose of calculating the observed colors, the intensity of the reflected light must be calculated for the three RGB components of the color model used, based on the ability of the surface to reflect light by means specific to the reproduced phenomena. Thus, the surface properties will be specified independently for the three RGB color components.

Therefore, the Phong lighting model is invoked as the approach of calculating the light and the material color of each object, so the equation is represented in formula 2.1 [23]

$$L_c = S_c + k_{dC} \sum_{i=1}^{n} f_{att}(r_i) E_{iC} N * I_i + k_{sC} \sum_{i=1}^{n} f_{att}(r_i) E_{iC} (I_i * O_s)^g + k_{aC} \qquad (2.1)$$

The meaning of each parameter :

– k$_{aR}$, k$_{aG}$, k$_{aB}$ - the reflection coefficients of ambient.

– k$_{dR}$, k$_{dG}$, k$_{dB}$ - the reflection coefficients of diffuse.

– k$_{sR}$, k$_{sG}$, k$_{sB}$ - the reflection coefficients of specular.

– g - coefficient of glosiness (shiness).

– S$_R$, S$_G$, S$_B$ - self luminance

– C - the character of color component, such as R - red, G - green, B - blue

– L$_c$ - luminance at the observer direction for the C component of RGB model.

– E$_{iC}$ - the intensity of i-th light for the C component.

– r$_i$ - distance to i-th light.

– N - unit surface normal vector.

– I$_i$ - unit vector to i-th light.

– O$_s$ - specularly reflected observer unit vector.

– A$_c$ - the intensity of ambient light.

– f$_{att}$(r)- light attenuation as a function distance.

The attenuation is located in interval (0,1), the formula related to this calculation is shown in formula 2.2

$$\text{f}_{att}(r_i) = min(\frac{1}{c_2r^2 + c_1r + c_0}, 1). \tag{2.2}$$

Due to the fact, that spotlight provides the lighting on the specific area of the scene. This area in which light hits has to be determined and then involves the calculation of lighting, otherwise to avoid darkness use the ambient light [20].

The main concept of the application is to link the program with shaders, where the variables received by uniform are used to calculate the matrices and Phong lighting. Moreover, there should be lights, otherwise, the scene would be dark, for this reason, the spotlights are involved.

Vertex specification is responsible for the stage where an application decides which shaders will take part in the program processing and setting up an ordered list of the vertices to deliver to the pipeline. Thus, the vertices are defined by boundaries of primitive. Primitives in general draw the shapes, in our case this is triangles. Moreover, this part concerns Vertex Array Objects (VAO) and Vertex Buffer Objects (VBO). VBO stores the actual vertex data itself, while VAO defines which data each vertex has. The possible set of them is a vertex, tessellation, and geometry shaders, as the reason that the vertex shader in pair with fragment one is enough to receive our expected results, we won't take into account other ones. All stages and their order are described in figure 2.5 [24].

The next step is the vertex shader, inside this process manipulates with the geometry, transformation to clip the Normalized device [17] space by producing Model View and Projection matrices. Moreover, the lighting should be included, otherwise, the scene would be fully dark, that's not satisfactory. In case the texturing is available, perform their operations to involve that.

The further stage is the Vertex Post-Processing, so as the shader-based vertex processing has finished then vertices go through a number of limited processing stages.

The rasterization step maps the resulting primitives to the corresponding pixels on the final window, resulting in fragments for the fragment shader to use. Moreover, as the fragment shader has not been executed the clipping process is invoked that discards all fragments that are outside user view, which leads to increased performance.

The fragment shader is involved to calculate the final color or the pixel. As the rule, the fragment shader contains the data regarding the 3D scene and there is evaluated the lights, shadows, the color of the light, etc.

The last stage named Per-Sample operations is used to collect data from fragment shader and pass through a sequence of steps, in the current project there is used only several of them, such as culling tests, depth tests, and color blending.

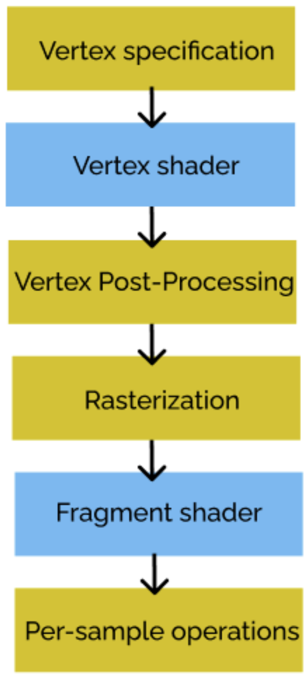


Fig. 2.5. Pipeline

### 2.5. POPULAR STANDARDS AND FORMATS

As computer graphics is gaining popularity, it leads to many ways how the data of 3D objects may be stored. The author would like to list below the most popular of them and then explain his decision. 3D files are those files that store all necessary data about 3D objects. Therefore, information that might be included in such files is following, geometry (shape of the model), appearance (color, materials, or texture), scene(camera, the light source position, other objects related to the 3D model), and animations (behavior of 3D objects). The last crucial part is that formats are divided into two categories, such as neutral and proprietary. The second type faces a problem in communication, for instance, let's assume that the same scene is modified in different environments as AutoDesk and Blender. They both would have different types as those systems are using their own propriety format that optimized for their software, AutoDesk using DWG file, while Blender has a Blend file, so they would not be reached between different software's. However, the neutral resolves this problem.

STL - this file format belongs to neutral ones and is one of the most popular for 3D printing unless it stores only the geometry information without any information about color or texture [12].

OBJ - also named as wavefront relates to the neutral group in ASCII variant, of the most popular interchange formats for 3D graphics, thanks to it its possible to save geometry together with color or texture information which is located in another file with an extension.MTL [7].

FBX - this type is related to a proprietary format. It allows animation, geometry, color, and textures. It is one of the most popular that is used in animation [3].

DAE - these extensions correspond to neutral file format, also known as COLLADA. It supports the storing appearance, geometry, animations, and scene. Moreover, this one allows the collection of data as physics and kinematics, only a few formats have such possibility. However, it became less popular due to its weakness to keep up with new technologies [2].

STEP - this file format is related to the neutral group. It provides information about geometry including its topology and geometrical tolerances, textures, and material types. [11]

Summary up everything described above and takes into account the table 2.4 is created.

| | STL | OBJ | FBX | DAE | STEP |
|---|---|---|---|---|---|
| category | proprietary | proprietary and neutral | proprietary | neutral | neutral |
| provides geometric data | yes | yes | yes | yes | yes |
| provides texture data | no | yes | yes | no | yes |
| provides animation data | no | no | yes | yes | no |
| provides color data | no | yes | yes | yes | no |
| ease of modification | yes | yes | no | no | yes |

Tab. 2.4. Table of comparing file formats.

The most convenient file format is OBJ, due to its simple structure, popularity, and ability to be extended. The possibility to be extended is one the most important parts, causing the idea to store two additional values, such as *instance* and *camera*. The instance is an approach to insert the object referenced to already existed in the scene, while the camera is responsible for the view position in the scene. In view of the fact that the simplest structure is the best option, another module should be able to generate the data and store it in an external extended file to further communication between modules. However, there is a problem with the .obj extension, such as it does not support the hierarchy. The Hierarchy is the collection of objects linked by parent/child relationships, in which transformations applied to a parent object are automatically passed on to its children. Thus, connecting several objects in a hierarchical chain allows you to create sophisticated animations [18]. Nevertheless, we do not look that far, as the transformation assigned to dynamic objects does not require the hierarchy. Moreover, this file format has to be extended and a full description of this is located in 3.8 section.

## 2.6. THIRD PARTY COMPONENTS

Substantially, we have accomplished a choice of OpenGL, so that supposes that now we are involved to select the component that would be able to invoke the implementation. Besides this basic OpenGL, in order to use it meaningfully, to combine it with the GUI window system that runs in the operating system. Moreover, to use these other additional operations, such as geometric operations, for example, with this OpenGL there is a set of other libraries that we should learn and that we should use.

— OpenGL - is a software interface to graphics hardware that provides a large set of interfaces, even over 250 different functions calls that may be involved to draw simple or complex 2D or 3D scenes from simple geometric primitives, for instance as points, lines, quads or more complex polygons.
— LWJGL - a Java library that supplies binding to a diversity of C libraries for graphics developing on Java, in order to contribute popular native APIs, such as OpenGL, Vulkan, OpenCL, etc. Moreover, the access is direct and ensures high performance.
— GLFW - is a multi-platform library for listed before APIs, that procures its own simple API for developing contexts or surfaces, collecting input and events, windows. In general, GLWF is written in C, in spite of that LWJGL has the direct communication between them and the library named lwjgl-glfw.
— Joml - is an OpenGL math library that concerns linear algebra operations, so it leads to satisfying any 3D application. In fact, it's also may be linked directly with LWJGL library.

— Lwjgl-opengl - is a library that supports all versions of OpenGL, as the reason the LWGJL library is evaluated into the solution as the main one with linking other relevant APIs, which were mentioned above.

The most popular libraries to create GUI are called *awt* and *swing*, so they are involved in the project to develop window-based applications i.

In case of assembly of video sequences was utilized the open-source program *ffmpeg* [4], by simple command it stores the video.

In addition, during the implementation processing were used tools, such as GitHub for version control, and JetBrains IntelliJ IDE Ultimate Edition, as the author is familiar with those and they provide all required instruments for successful development.

## 2.7. SUMMARY

To summarise what has been said in this chapter, treating the authoring module as a stand-alone or plug-in for integration into the overall system, corner cases have been considered. Therefore, through analysing other similar solutions, it was necessary to consider typical mistakes that should be avoided at the beginning of the project configuration and, if possible, during the implementation process. Thus, in the end, the system should be with GPU-CPU cooperation to improve the efficiency of the application. The most adaptive technologies, file exchange formats and environments are known and will be used during development. Moreover, the visualizer implemented by the author will not be the same as the thousands already existing. Thus, unique features will be leveraged.

# 3. DESING ELEMENTS

## 3.1. REQUIREMENTS

Well-designed and well-documented requirements are significant to any successful software development project. There are two basic types of system requirements that should be collected by those working on software projects. System requirements can be divided into functional and non-functional requirements. The user of the application is person who will experiment with whole simulation for projection of animation with dynamic camera. The requirements for the rendering module come from the aims and requirements of the whole animation.

### 3.1.1. Functional requirements

1. The system should be able to handle a specific file format (extended obj file).
2. The system should be able to handle the material libraries, for example mtl files.
3. The system should be able to render the context of given files.
4. The system should be able to set the position of Camera.
5. The system should be able to draw instances.
6. The system should be able to define static objects.
7. The system should be able to define dynamic objects.
8. The system should be able to store the result as an image.
9. The system should be able to store the result into a video.
10. The system should be able to create the scene with built-in renderer.
11. The system should be able to create the scene with alternative render.
12. The system should be able to render the scene with spotlight illumination.
13. The system should be able to display shadow effect.
14. The system should be able to display weather effect.
15. The system should be able to support 2D texturing.
16. The system should be able to provide all features related to GUI.
17. The system should be able to handle external file formats, if relevant converter is provided.

### 3.1.2. Non-functional requirements

1. Windows OS.
2. The Application should be implemented as an install-able program (not as e.g. web service).

3. The period of data storage in the system as long as user want it.
4. The PC should have graphic card, more powerful would provide more performance.
5. The application should produce 10 FPS, in case of not complex scene.
6. The application should work without internet connection.

### 3.2. SPECIFICATION OF USE CASES

A use case specification captures requirements, usually for a system, in the form of a use case, which contains descriptive steps of requirements in a logical sequence, so that the document specification can be understood by users to get a signature of their requirements, and by testers and developers to understand what the system needs to test and build the system functionality detailed in the system use case.

**Use case 1 : Display the scene**

**Basic flow:**

1. A User would like to view the rendered scene.
2. The User imports the required files, such as extended obj and mtl.
3. The application provides the possibility of selecting the renderer.
4. The application processes current files.
5. The application after handle process displays the whole scene.

**Alternative flow:**

2a. The user would like to import files from another file format. For this purpose, it assumes that the user is qualified enough and the converter prepared by him is invoked. Thus, the execution should be performed through a batch script.

**Exception flow: Incorrect converter**

2b. The converter provided by the user does not convert data correctly.
4b. The application verifies that the files imported are incorrect.

**Use case 2 : Store the scene view**

**Basic flow:**

1. A user would like to store the rendered scene.
2. The user configure the application and scene displayed.
3. The user selects the approach of storing a single image.
4. The user selects the name and folder where the content would be saved.
5. The application successfully stores data.

**Alternative flow:**

3a. The user selects the approach of storing a video.

**Exception flow: The video could not be created**

3b. There are less than 2 frames.

## 3.3. ARCHITECTURE

Software architecture briefly is the organization of a system. This organization includes all the components, how they interact with each other, the environment in which they work, and the principles used in developing the software. One of the goals is to achieve successful integration between modules to produce one huge application, instead of 4 independent subprograms. Therefore, the following approach was taken into account : research the project type, establish goals and objectives, collect relevant information, identify strategy, quantify the requirements, summarize the program. The further subsections would describe the interaction between modules and author's one.

### 3.3.1. Architecture of the author system

As it was mentioned in the previous chapter, the environment in which the program would be developed is Java. Thus, the program was created as a maven project with dependencies described in third-party components. As one of the main purposes of the application is to provide high-performance focus related to CPU-GPU cooperation. In view of the fact that the architecture of OpenGL is based on a client-server model. An application program written to use the OpenGL API executes on the CPU. The implementation of the OpenGL graphics engine in communication with the GLSL shader program executes on the GPU [14].

Modeling, rendering, and interaction is very much a cooperative process between the CPU and GPU programs written in GLSL. An important part of the design process is to decide how best to divide the work and how best to package and communicate required information from the CPU to the GPU. The approach of communication is shown in figure 3.1

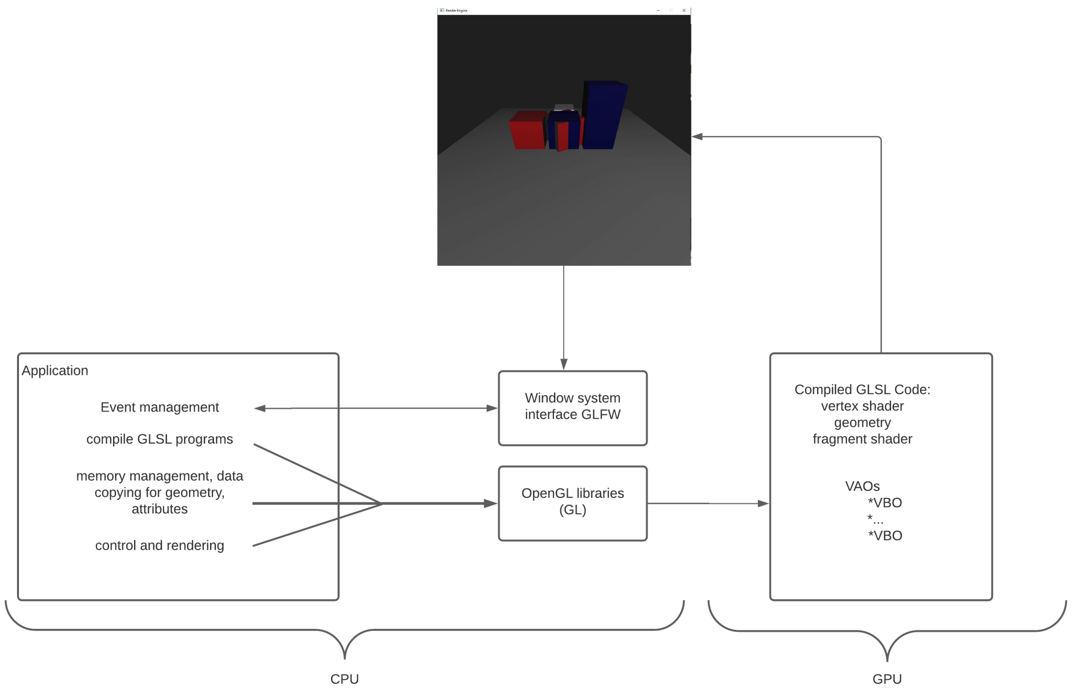
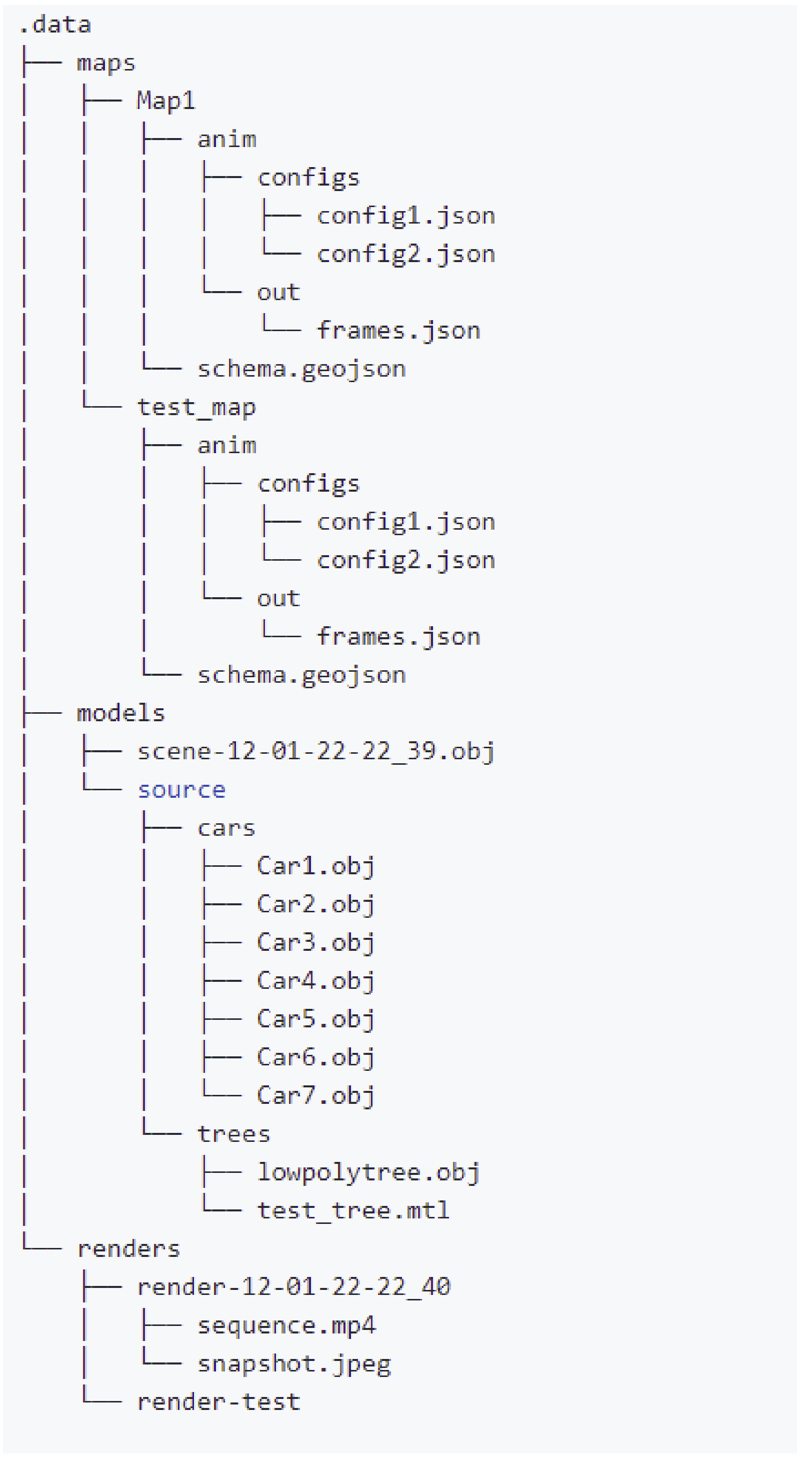

Application
Event management
compile GLSL programs
memory management, data copying for geometry, attributes
control and rendering
Window system interface GLFW
OpenGL libraries (GL)
Compiled GLSL Code: vertex shader geometry fragment shader
VAOs
*VBO
*...
*VBO
CPU
GPU

Fig. 3.1. Architecture of author's module

### 3.4. DIAGRAMS DESCRIBING INTERACTION WITH EXTERNAL MODULES

This chapter describes the exchange between modules of the huge system. Thus, to provide a working application where each successive module can exist, the project tree structure shown in the figure 3.2 has been designed to store all necessary data after each module has been executed, ensuring that the previous module can supplement the subsequent one and generate information for the subsequent ones. In the case of the last stage, called rendering and related to the author's work, this module does not generate further data, but it is given more responsibility as it describes the end-to-end interaction.

Briefly summarising, the tree structure describes how the data of exchange between modules is located inside a complex program.



```
.data
├── maps
│   ├── Map1
│   │   ├── anim
│   │   │   ├── configs
│   │   │   │   ├── config1.json
│   │   │   │   └── config2.json
│   │   │   └── out
│   │   │       └── frames.json
│   │   └── schema.geojson
│   └── test_map
│       ├── anim
│       │   ├── configs
│       │   │   ├── config1.json
│       │   │   └── config2.json
│       │   └── out
│       │       └── frames.json
│       └── schema.geojson
├── models
│   ├── scene-12-01-22-22_39.obj
│   └── source
│       ├── cars
│       │   ├── Car1.obj
│       │   ├── Car2.obj
│       │   ├── Car3.obj
│       │   ├── Car4.obj
│       │   ├── Car5.obj
│       │   ├── Car6.obj
│       │   └── Car7.obj
│       └── trees
│           ├── lowpolytree.obj
│           └── test_tree.mtl
└── renders
    ├── render-12-01-22-22_40
    │   ├── sequence.mp4
    │   └── snapshot.jpeg
    └── render-test
```

Fig. 3.2. Project tree structure

According to the technique selected by the author, the internal data structure should be organized and the form of these data must be compatible with expectations of the OpenGL context. Therefore, the listed below data types are defined.

**HashMap** as the collection of instances and its properties, such as position, rotation, scale and material. Every instance referencing into the object called mesh

**ArrayList** as the collection of the all objects from extended obj file.

**Vector3f** as the vector that contains of x,y,z values, used for storing the parameters required for matrices, for instance the transformation.

**Vertex** as the set of vectors, such as position, textures and normals.

**FloatBuffers** used as the memory allocation of vertices, textures, and normals, also contains material information. Buffers must be created and filled before rendering.

**Matrix4f** used as the transformation operation for calculating the final values of the following matrices : translation, rotation, scaling, model, view, projection [22].

Before we moved to the matrices calculation, the crucial note, that homogeneous coordinates [19] are widely used in computer graphics because they allow common vector operations, especially for transformation and perspective projection to be represented as a matrix by which a vector is multiplied.

Translation :

$$\begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{bmatrix} x + X * 1 \\ y + Y * 1 \\ Z + Z * 1 \\ 1 \end{bmatrix}$$

Rotation around X-axis :

$$\begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{bmatrix} cos\theta * x + sin\theta * z \\ y \\ -sin\theta * x + cos\theta * z \\ 1 \end{bmatrix}$$

Regarding rotation, there author faced with a problem, since by default it rotates around origin not its own center, so for resolving this problem the decision was taken as firstly translate them into 0,0,0 coordinates (origin) and then perform transformation calculations in following order : rotation, scaling and translation to its final destination [15].

Scaling :

$$\begin{bmatrix} SX & 0 & 0 & X \\ 0 & SY & 0 & Y \\ 0 & 0 & SZ & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{bmatrix} SX * x \\ SY * y \\ SZ * z \\ 1 \end{bmatrix}$$

Applying of translation, rotation and scaling matrices elaborates to model matrix. The view and projection matrix are defined below [21] :

View matrix - the usage is transformation from world coordinates to observer coordinates. Creates with the lookAt function that accepts 3 vector3f parameters, such as eye, target and vertical direction.

Projection matrix - the perspective of projection matrix is set in a symmetric way. For this approach it requires following parameters, fov, aspect, znear, zfear.

*fov* - viewing angle on plane.

*aspect* - the ratio of width/height of the image.

*znear* and *zfear* - location of clipping planes along z axis.

Thus, with configuration of those matrices we might observe the objects on the plane.

### 3.6. ACTIVITY DIAGRAM

Based on the list of requirements mentioned above, the following diagtam was developed by the author as shown in figure 3.3 The selected approach by author supports for efficient way of displaying frame by frame, as the static object are stored and do no need to rewrite the same information each time, that leads to increasing the performance of application.

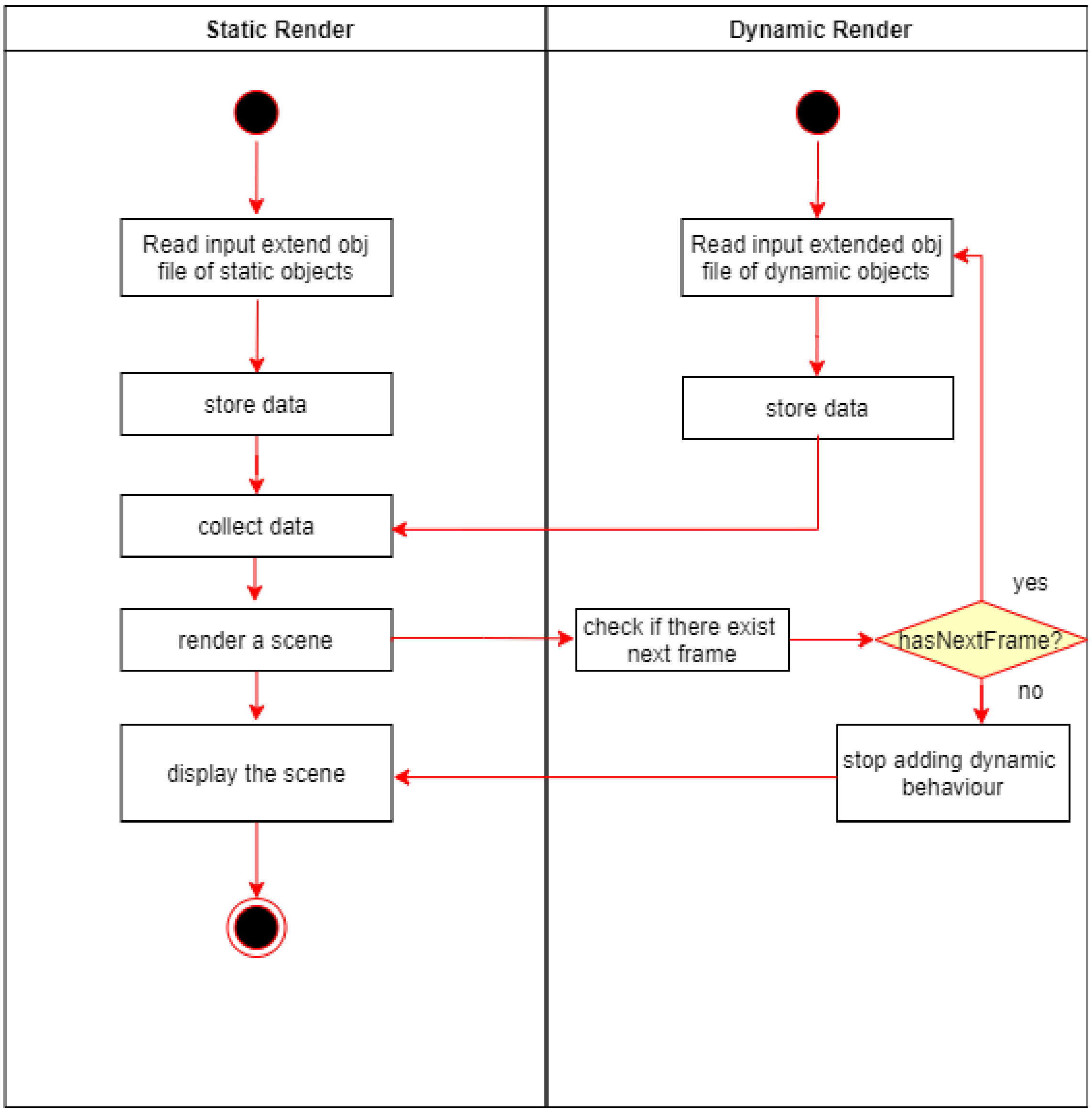

Static Render
Dynamic Render
Read input extend obj file of static objects
store data
collect data
render a scene
display the scene
Read input extended obj file of dynamic objects
store data
check if there exist next frame
hasNextFrame?
yes
no
stop adding dynamic behaviour

Fig. 3.3. Activity Diagram

### 3.7. CLASS DIAGRAMS

The diagram shown in figure 3.4 contains the most important classes of the project. First of all, the MeshObjectLoader is invoked to read the input files and store data into relative sets. Due to fact that the program takes as input extended obj file, meshes stores as a unique one and then by involving instances, it refers to already existing index. Each instance of mesh contains the data set that is divided into parts and calculated in the relevant classes. Some of them are Vertex, Transformation, Material classes. The ShaderProgram class is responsible for linking the GLSL code, more precisely Vertex and Fragment shaders. The shader program is called inside Scene class that collects all data about the frame which would be displayed on Window. There are various methods that were implemented, however, there are some which are crucial parts. The method named objLoader() located in the class called MeshObjectLoader is responsible for handling all data received by extended obj file, especially the values which do this file extended, instance, and camera. Thus, the mesh class store this data, which leads to the scene weight. Moreover, not less important the method called getBounds() finds the min and max values of vertices coordinates and then shift them to the origin, which leads to successful transformation matrices operations. Therefore, should be calculated the model, view, and projection matrices for finally display scene, and of course to place on the plane that could be obtained by camera properties. Thus, the scene starts the rendering process based on the information of static and dynamic geometry objects, material color, light properties, extended values, and a shader program that links the vertex and fragment shaders.

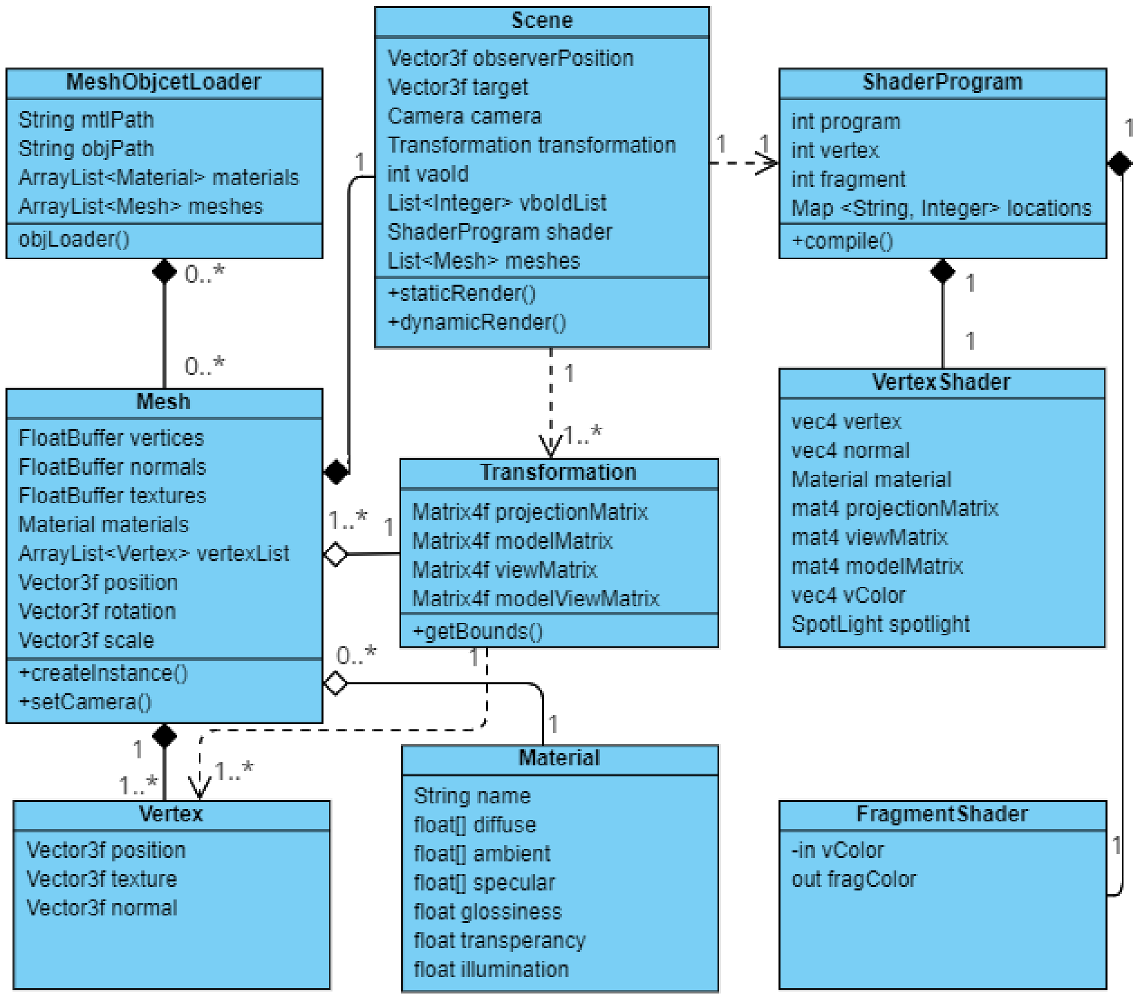


Fig. 3.4. Uml Diagram

For memory management on GPU side the shader program is developed. This program manages buffers that are created in the main program, such as, vertex, normal and material ones are involved. Moreover, other required parameters are send by uniform call. Therefore, with this provided data the further calculation are performed, computing position in NDC space, calculation of spotlights, Phong lighting model and color which is sent to the fragment shader.

## 3.8. EXTERNAL DATA FORMATS

As the external data the module accepts the extended obj 3.1 and mtl 3.2 file. The structure is extended by two additional values named as instance and camera.

First one providing the "instance" and its parameters are referenced to object id, transformation matrices, such as position, rotation, scale and material id, while the second one "camera" is in charge of view position in the scene. In case that, camera field is empty, the default values would be used. In case of instances are empty or do not exist, the default scene will be displayed.

| | |
|---|---|
| mtllib name.mtl | First line of file always saves name of mtl to use |
| o object name | Object name changes after change of object type |
| v x y z | Vertex information stored in this way |
| vn x y z | Calculated vector normal |
| usemtl Road | Smooth shading is turned off in our models |
| s off | surface is turned off |
| f vId//vnId vId//vnId viId//vnId | Faces are stored in such a way. vId - vertex ID, vnId - vertice normal ID |
| instance | objId dx dy dz rx ry rz sx sy sz materialId |
| camera | x1 y1 z1 for observer x1 y1 z1 for target and angle |

Tab. 3.1. Table of obj variables,

| | |
|---|---|
| newmtl Ground | Defines material named ‘Ground’ |
| Ns 500 | Specular exponent |
| Ka 0.8 0.8 0.8 | The ambient color of the material |
| Kd 0.8 0.8 0.8 | The diffuse color |
| Ks 0.8 0.8 0.8 | The specular color |
| d 1 | Material transperancy |
| illum 2 | Illumination of the material |

Tab. 3.2. Table of extended obj variables,

## 3.9. UI DESIGN

The author sees fit to create interfaces which users find easy to use and pleasurable. Graphical user interface (GUI) has to be as much as possible simplified for users who would see the application first time. Thus, a simple wireframe of all needed buttons and blue-print of application with shown in 3.5 indicates all needed buttons. There is presented two menus with options for the user, such as menu with following operations : import, save as an image, save as video, quit the application and render selection. Those operations would be enough to display the scene with possibility of storing the result.

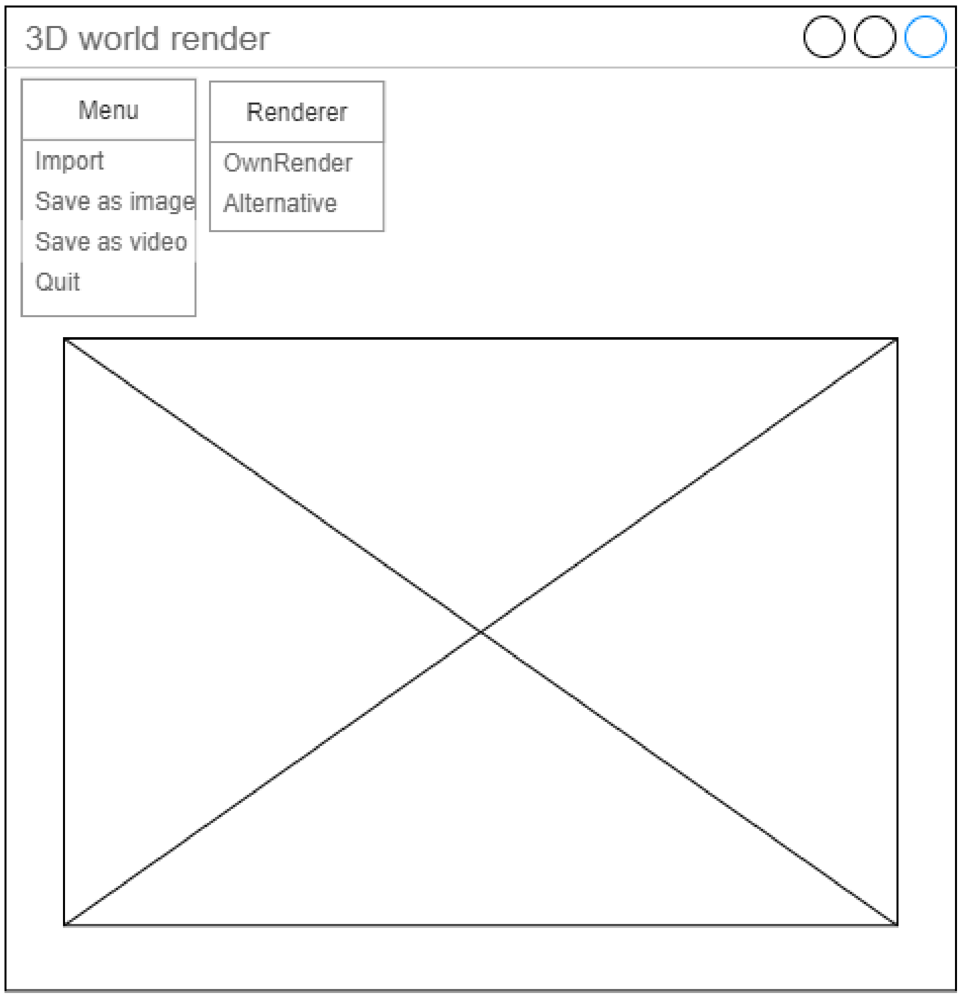


Fig. 3.5. Wireframe of desktop application

# 4. INSTALLATION AND USER MANUAL

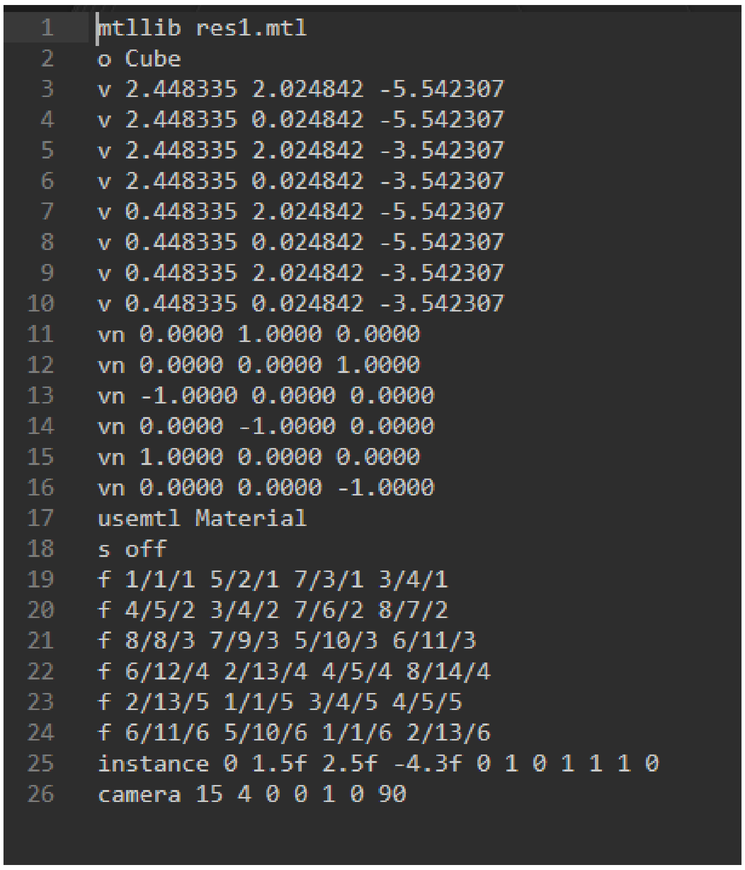
## 4.1. INSTALLATION

As for the installation it is enough to have installed Java Runtime Environment (JRE) and manually run the Render.exe file that contains of all required jar files and immediately opens the UI program on which the user is able to select actions he would like to.

## 4.2. DESCRIPTION OF ALL UI ELEMENTS

User interface (UI) elements are the blocks we use to build applications or websites. They add interactivity to the user interface by providing touchpoints for users to navigate the site: buttons, an application bar, menu items, error messages, and checkboxes.

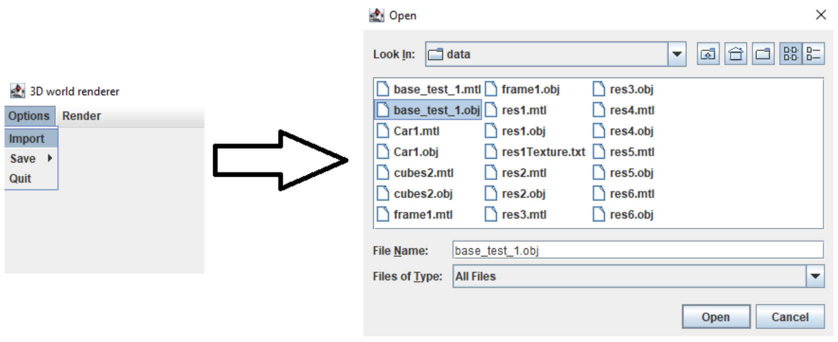
First off all, the file(s) should be configured that provides the possibility to import them. To verify correctness of the file, it should have the same structure as shown in figure 4.1



```
mtllib res1.mtl
o Cube
v 2.448335 2.024842 -5.542307
v 2.448335 0.024842 -5.542307
v 2.448335 2.024842 -3.542307
v 2.448335 0.024842 -3.542307
v 0.448335 2.024842 -5.542307
v 0.448335 0.024842 -5.542307
v 0.448335 2.024842 -3.542307
v 0.448335 0.024842 -3.542307
vn 0.0000 1.0000 0.0000
vn 0.0000 0.0000 1.0000
vn -1.0000 0.0000 0.0000
vn 0.0000 -1.0000 0.0000
vn 1.0000 0.0000 0.0000
vn 0.0000 0.0000 -1.0000
usemtl Material
s off
f 1/1/1 5/2/1 7/3/1 3/4/1
f 4/5/2 3/4/2 7/6/2 8/7/2
f 8/8/3 7/9/3 5/10/3 6/11/3
f 6/12/4 2/13/4 4/5/4 8/14/4
f 2/13/5 1/1/5 3/4/5 4/5/5
f 6/11/6 5/10/6 1/1/6 2/13/6
instance 0 1.5f 2.5f -4.3f 0 1 0 1 1 1 0
camera 15 4 0 0 1 0 90
```

Fig. 4.1. Simple Example of extended obj

As the at least one file is ready to be displayed, we have to import the stuff by proceeding the import button in menu bar named as Options. The import process presented in figure 4.2



Look In: data
File Name: base_test_1.obj
Files of Type: All Files

Fig. 4.2. Import button

After successful import the user might select the renderer, for example firstly was selected the author's one. The following window is opened as shown in figure 4.3

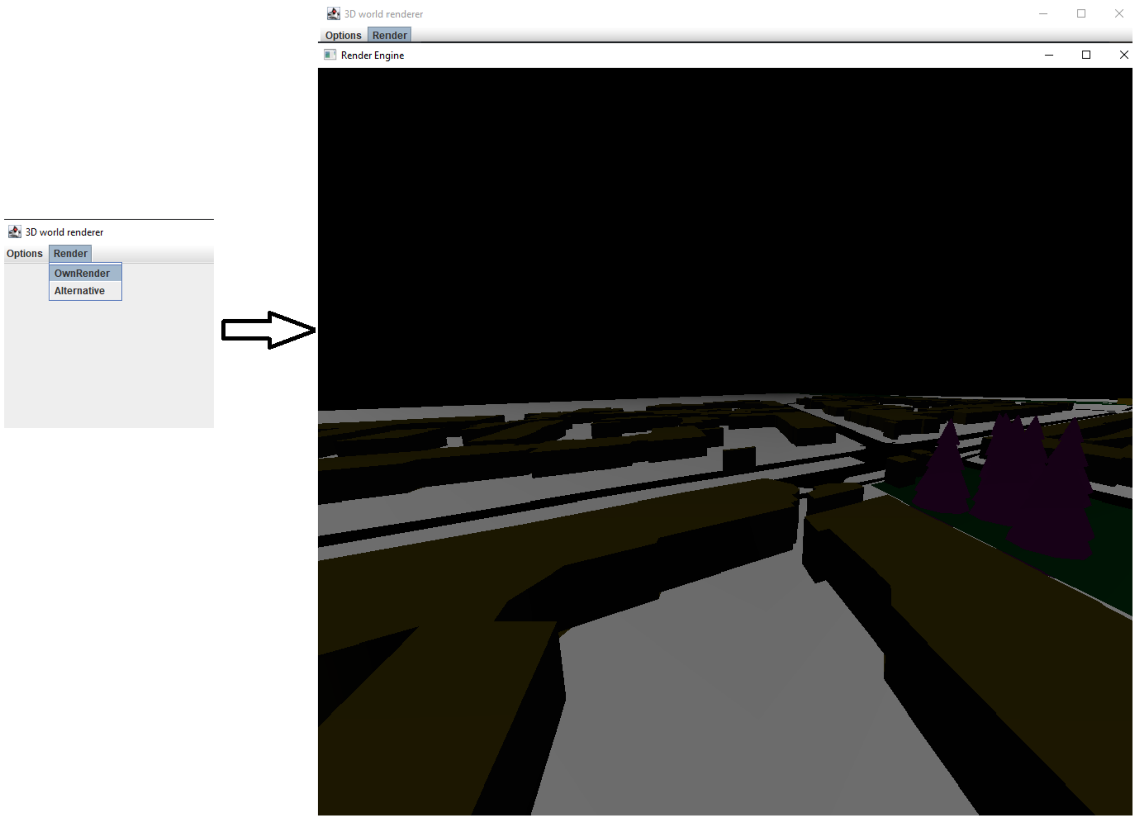


Fig. 4.3. Author's Render

After successful import the user might select the renderer, for example secondary was selected the alternative's one. The following window is opened as shown in figure 4.4

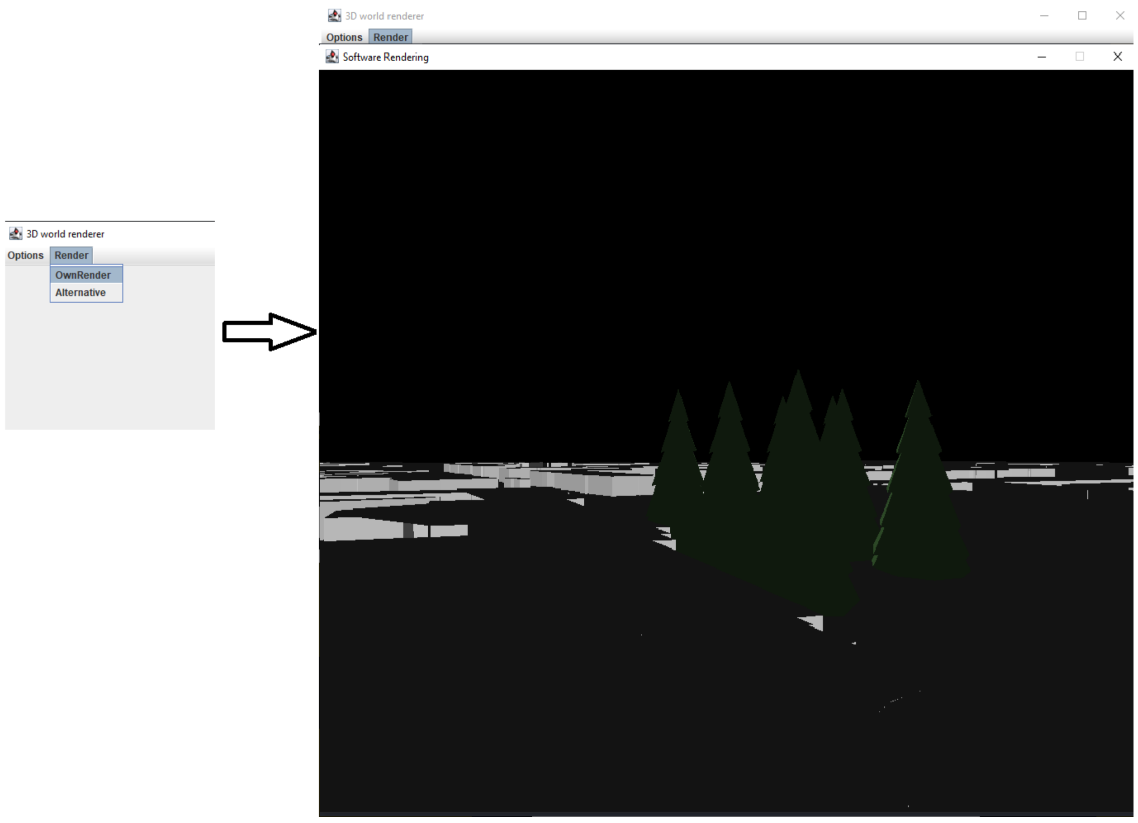
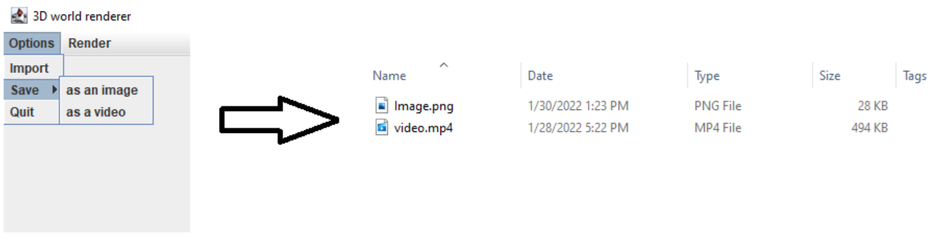


Fig. 4.4. Alternative Render

The user would like to store the result as image or video, then he should press save button and select the format as shown in figure 4.5



| Name | Date | Type | Size | Tags |
|---|---|---|---|---|
| Image.png | 1/30/2022 1:23 PM | PNG File | 28 KB | |
| video.mp4 | 1/28/2022 5:22 PM | MP4 File | 494 KB | |

Fig. 4.5. Save Content

Quit button is responsible to force application closing.

The case when dynamic objects are existing in the scene the following behaviour presented in figure presented below 4.6, might be viewed. Thus, the alternative render shown in figure 4.7

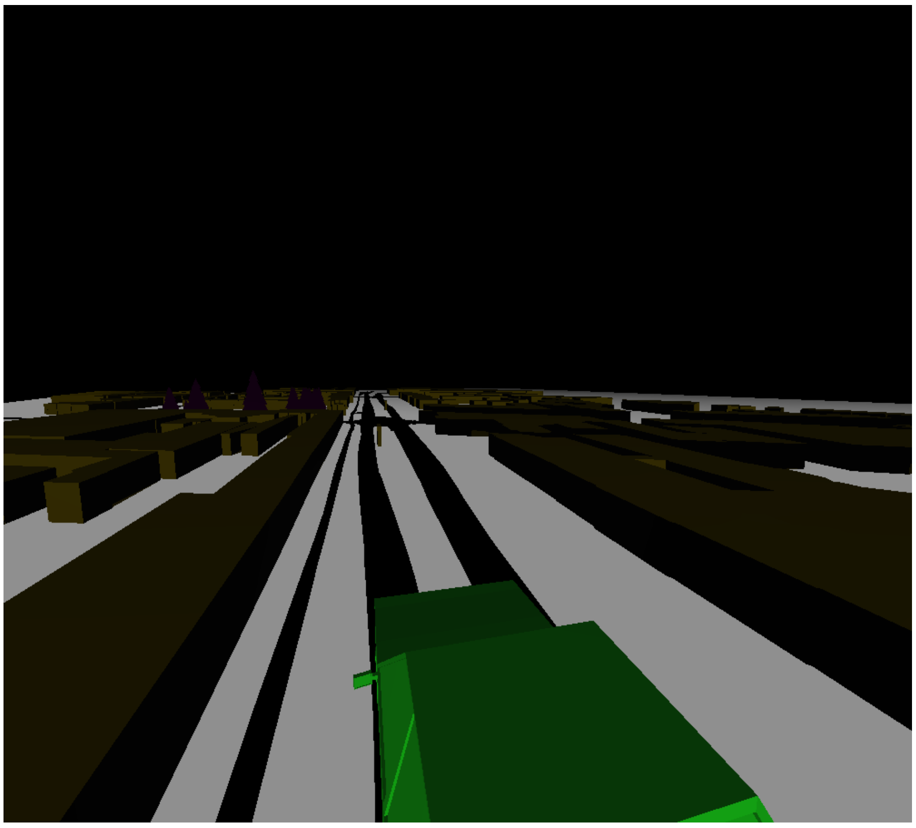


Fig. 4.6. Video of built-in renderer

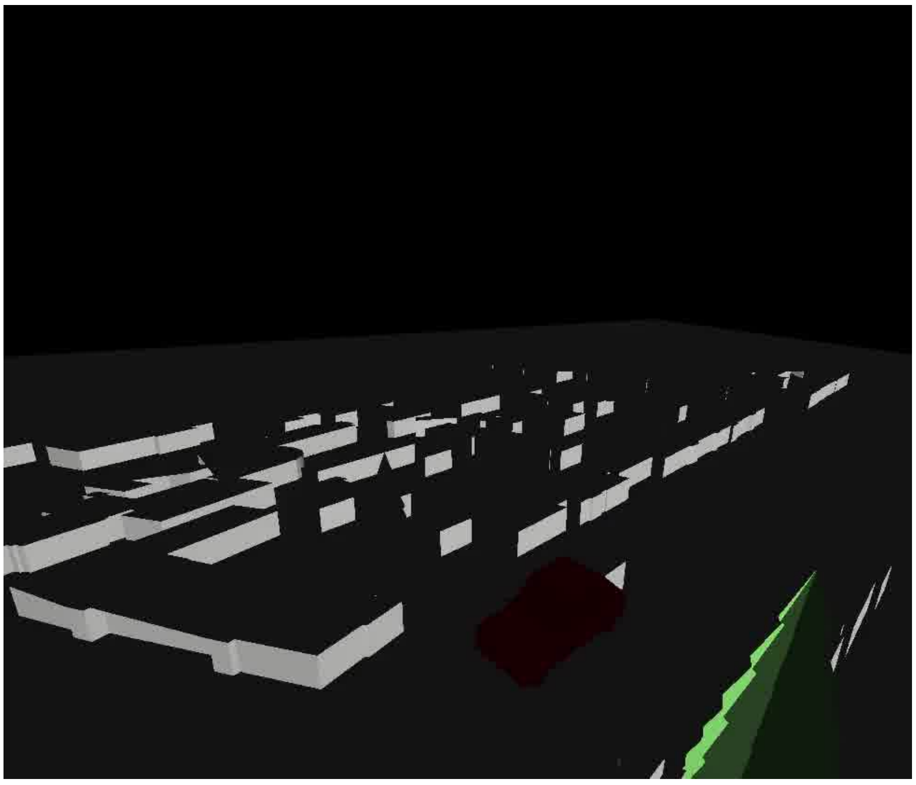


Fig. 4.7. Video of alternative renderer

# 5. TESTING

The testing stage is a crucial part of the any applications, as the rule it unlocked some unexpected behaviours of the program, thankful that the developers are able to fix their mistakes and improve the quality of the whole system. That's why the application was tested to verify its correctness. There are many ways how to test the environment, starting from manual testing and finish up integration ones.

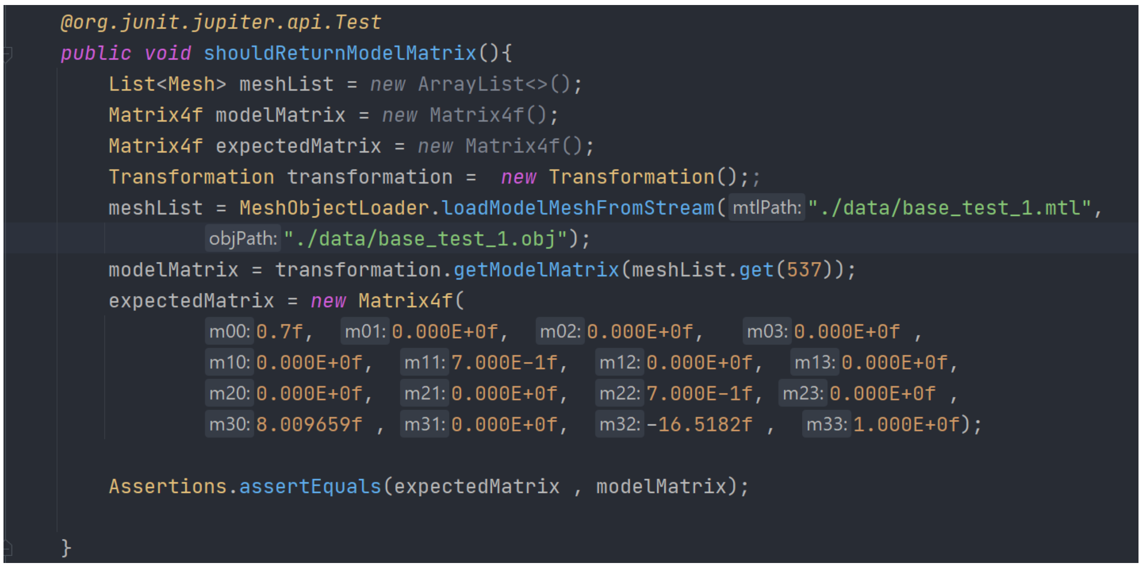
On one hand the manual testing is not the proper case for testing, as it requires creation of a configuration and manually undergo through each steps.

On the other hand automatics tests are executed by the machine that evaluates ...

## 5.1. UNIT TESTING

Unit testing conducted as low-level features, as they are close to the source of the application. Therefore, it's based on testing single methods and functions of the classes, modules or even components used by the software. In general, the unit testing is fairly cheap to automate.

The example of model Transformation testing for specific object as described in figure 5.1

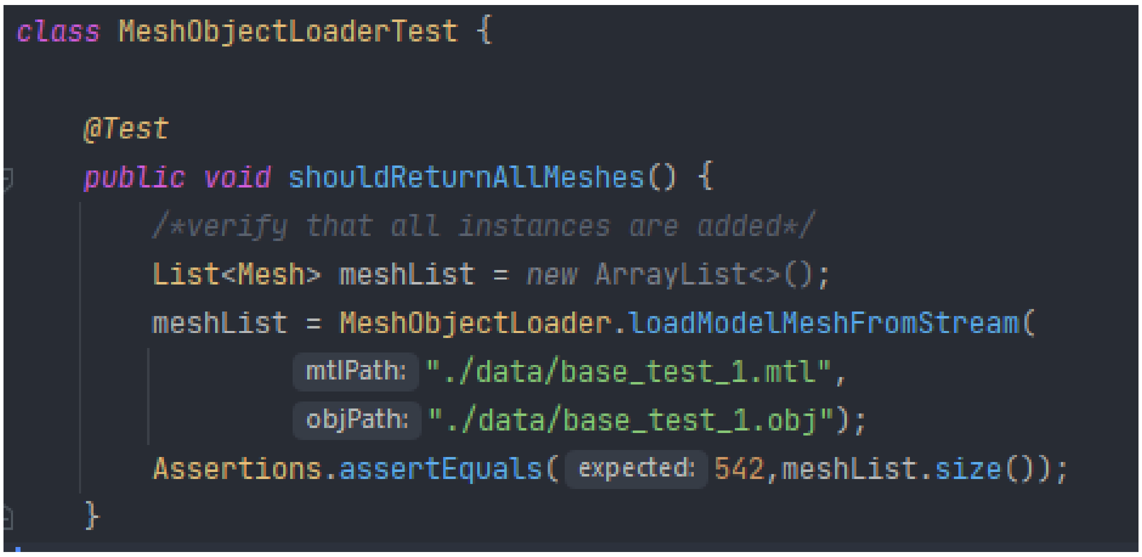


```java
@org.junit.jupiter.api.Test
public void shouldReturnModelMatrix(){
    List<Mesh> meshList = new ArrayList<>();
    Matrix4f modelMatrix = new Matrix4f();
    Matrix4f expectedMatrix = new Matrix4f();
    Transformation transformation =  new Transformation();;
    meshList = MeshObjectLoader.loadModelMeshFromStream( mtlPath: "./data/base_test_1.mtl",
            objPath: "./data/base_test_1.obj");
    modelMatrix = transformation.getModelMatrix(meshList.get(537));
    expectedMatrix = new Matrix4f(
            m00: 0.7f,  m01: 0.000E+0f,  m02: 0.000E+0f,   m03: 0.000E+0f ,
            m10: 0.000E+0f,  m11: 7.000E-1f,  m12: 0.000E+0f,  m13: 0.000E+0f,
            m20: 0.000E+0f,  m21: 0.000E+0f,  m22: 7.000E-1f,  m23: 0.000E+0f ,
            m30: 8.009659f ,  m31: 0.000E+0f,  m32: -16.5182f ,  m33: 1.000E+0f);

    Assertions.assertEquals(expectedMatrix , modelMatrix);

}
```

Fig. 5.1. Unit testing example

## 5.2. INTEGRATION TESTING

As the module interacts with another ones only by reading the files from another module which is responsible for generalization of 3d files, such as extended obj and mtl there were produced test for verifying reading process of those files. Below shown in figure 5.2 one exemplary case to check if all instances were successfully added to render process.



```
class MeshObjectLoaderTest {

    @Test
    public void shouldReturnAllMeshes() {
        /*verify that all instances are added*/
        List<Mesh> meshList = new ArrayList<>();
        meshList = MeshObjectLoader.loadModelMeshFromStream(
                mtlPath: "./data/base_test_1.mtl",
                objPath: "./data/base_test_1.obj");
        Assertions.assertEquals( expected: 542,meshList.size());
    }
```

Fig. 5.2. Integration exemplary test

## 5.3. PERFORMANCE TESTING

The performance was checked on author's laptop with the following characteristics :

- Intel(R) Core(TM) i7-7700HQ CPU 2.8 GHz
- RAM 16 GB
- gpaphic card GEFORCE GTX 1050
- Windows 10 x64
- 40% free memory

All possible outcomes are evaluated and transformed into the table 5.1 :

| | Before execution (%) | Only static (%) | Only dynamic (%) | Both (%) |
|---|---|---|---|---|
| CPU | 11 | 33 | 37 | 49 |
| GPU | 2 | 8 | 10 | 12 |
| RAM | 49 | 55 | 51 | 63 |

Tab. 5.1. Table of performance per CPU, GPU, RAM.

By summary up, it can be noticed that performance, beats the expectations. Moreover, the one of the most crucial aims is achieved, by executing the program not only CPU side takes the loading, but the GPU as well.

# 6. CONCLUSIONS

The engineering solution developed by the author fulfills the requirements for the visualization of the 3D world. The approximate amount of written codes is 500 lines, the author was trying to stick to SOLID and Clean code principles. The application might be improved in the future by further developers with providing additional features. Due to complex of the program, there was not implemented all functional requirements. Moreover, other modules were not fully completed, resulting in the loss of some test data, which requires tougher testing.

The author has achieved the following goals:

— Reading the scene from extended obj file.
— Reading the material color from extended obj file.
— Dynamic camera position.
— The possibility to activate an alternative render.
— Single frame rendering with OpenGL mode with the following capabilities :
  - Phong lighting model;
  - camera specification by eye position, viewing direction, horizontal field of view angle and aspect ratio;
  - mesh geometry rendering;
  - spot lights illumination;

— Storing rendered frames as separated images.
— Assembling animation sequence and storing it as a movie file.
— GUI management for all configurable elements (renderer selection, image/video storage management).

## 6.1. FUTURE RECOMMENDATIONS

Some performance improvements relative to the render features, converter for other file formats, and another operating systems in which application might be executed could be involved to improve the current solution.

One of the most obvious parts that might be included in the implementation to display a more realistic view is the supporting additional effects, such as shadow, weather, etc. The solution proposed by the author might accept such extensions, for this purpose, it requires understanding the concept of the application. This could be developed by providing the depth map, as it is texture rendered from the light's perspective that would be used in testing for shadows.

Another possibility to improve the presented solution is to develop a multi-tasking converter which would be able to handle various file formats.

At this time the application might be executed only on Windows, nevertheless, it could be a good point to extend the application on other operating systems, for example, Linux and macOS.

## BIBLIOGRAPHY

[1] *AirSim*, https://microsoft.github.io/AirSim/. [Online; Last Opened 4 January 2022].

[2] *Dae documentation*, https://docs.fileformat.com/3d/dae/. [Online; Last Opened 20 January 2022].

[3] *Fbx documentation*, https://docs.fileformat.com/3d/fbx/. [Online; Last Opened 20 January 2022].

[4] *ffmpeg*, https://ffmpeg.org/. [Online; Last Opened 4 January 2022].

[5] *Lumion*, https://lumion.com/. [Online; Last Opened 4 January 2022].

[6] *LWJGL*, https://www.lwjgl.org/. [Online; Last Opened 4 January 2022].

[7] *Obj documentation*, https://docs.fileformat.com/3d/obj/. [Online; Last Opened 20 January 2022].

[8] *OpenGL*, https://www.opengl.org/. [Online; Last Opened 4 January 2022].

[9] *Pov-Ray*, http://www.povray.org/. [Online; Last Opened 4 January 2022].

[10] *PyOpenGL*, http://pyopengl.sourceforge.net/documentation/index.html. [Online; Last Opened 4 January 2022].

[11] *Step documentation*, https://docs.fileformat.com/3d/step/. [Online; Last Opened 20 January 2022].

[12] *Stl documentation*, https://docs.fileformat.com/cad/stl/. [Online; Last Opened 20 January 2022].

[13] *WebGL*, https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API. [Online; Last Opened 4 January 2022].

[14] Angel, E., Shreiner, D., *An introduction to shader-based opengl programming*, w: *ACM SIGGRAPH 2009 Courses* (2009), str. 1–152.

[15] Angel, E., Shreiner, D., *Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL*, 6th wyd. (Addison-Wesley Publishing Company, USA, 2011).

[16] Cusin, G., Dvorkin, I., Pitrou, C., Uzan, J.P., *Properties of the stochastic astrophysical gravitational wave background: Astrophysical sources dependencies*, Physical Review D. 2019, tom 100, 6.

[17] Hughes, J.F., van Dam, A., McGuire, M., Sklar, D.F., Foley, J.D., Feiner, S., Akeley, K., *Computer Graphics: Principles and Practice*, 3 wyd. (Addison-Wesley, Upper Saddle River, NJ, 2013).

[18] KEHOE, C., STASKO, J., TAYLOR, A., *Rethinking the evaluation of algorithm animations as learning aids: an observational study*, International Journal of Human-Computer Studies. 2001, tom 54, 2, str. 265–284.

[19] Klawonn, F., *Introduction to Computer Graphics: Using Java 2D and 3D (Undergraduate Topics in Computer Science)*, 1 wyd. (Springer Publishing Company, Incorporated, 2008).

[20] McReynold, T., *Advanced Graphics Programming Using OpenGL.* (Morgan Kaufmann).

[21] Mehta, P., *Learn OpenGL ES: For Mobile Game and Graphics Development* (Apress, 2013).

[22] Overvoorde, A., *Modern OpenGL Guide test* (https://github.com/Overv/Open.GL, 2019).

[23] Sas, J., *Advanced computer graphics : practical advanced computer graphics - laboratory assignments*. 2011.

[24] Shreiner, D., Group, T.K.O.A.W., *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*, 7th wyd. (Addison-Wesley Professional, 2009).