

Czech Technical University in Prague

Faculty of Electrical Engineering



Master's thesis

Parametric Model of MPII Building

Bc. Ondřej Linda

Thesis supervisor: Ing. Vlastimil Havran, Ph.D.

Study program: Electrical Engineering and Information Technology

Field of Study: Computer Science and Engineering

Study Specialization: Computer Graphics

May 2010

Acknowledgments:

First, I would like to acknowledge my family and friends for their continuous support during my studies. Next, I would like to thank my thesis supervisor Dr. Vlastimil Havran for guidance and help during my work on this thesis. Furthermore, I would like to appreciate Josef Zajac and Jiří Drahokoupil for their previous work on this project.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady uvedené v příloženém seznamu literatury.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona c.121/2000Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 15. května 2007

.....

Abstract

The scope of this master thesis is to study predictive image synthesis. Predictive image synthesis is a part of photo-realistic rendering, which is based on using physically exact algorithms for computing global illumination. The prediction of the final images is based on a real world model, thus the results can be objectively evaluated.

The aim of this work was to extend the previously created 3D model of the MPII building, which consisted of building geometry, approximated materials and measured light characteristics. This static model of the MPII building was parameterized, allowing the following dynamic modifications of the model: i) geometry modifications of movable elements such as doors, windows or window shutters, ii) controlling the lights in the building, iii) modifying the cameras in the virtual scene. Further, a user interface was implemented, which allowed model animations and generation of input data for a physically based rendering algorithm.

The correctness of the implemented parametric model of the MPII building was verified by creating five reference video sequences of the animated model.

Abstrakt

Tato diplomová práce se zabývá prediktivní syntézou obrazu. Prediktivní syntéza obrazu je součástí foto-realistického renderingu, který je založen na používání fyzikálně přesných algoritmů pro výpočet globálního osvětlení. Predikce vzhledu renderovaného obrázku je provedena na základě reálného modelu a výsledek může být objektivně vyhodnocen.

Cílem této diplomové práce bylo rozšířit dříve vytvořený 3D model budovy MPII, který obsahoval geometrii budovy, aproximované materiály a změřené světelné zdroje. Tento statický model budovy MPII byl parametrizován, což umožnilo následující dynamické modifikace modelu: i) nastavení pohyblivých objektů budovy jako jsou dveře, okna a žaluzie, ii) nastavení světel v budově a iii) nastavení kamer pro náhled scény. Dále byla implementována možnost animace parametrizovaného modelu a vygenerování vstupních dat pro fyzikálně přesný renderovací algoritmus.

Korektnost implementovaného parametrizovaného modelu budovy MPII byla ověřena tvorbou pěti referenčních videí.

Contents

1 INTRODUCTION	- 1 -
1.1 Predictive Image Synthesis	- 1 -
1.2 Previous Work	- 1 -
1.3 Current Work	- 2 -
1.4 Thesis Structure	- 3 -
 2 IMAGE SYNTHESIS	 - 5 -
2.1 Predictive Image Synthesis	- 5 -
2.2 Rendering Equation	- 7 -
2.2.1 Radiometry	- 8 -
2.2.2 Bidirectional Reflectance Distribution Function	- 10 -
2.2.3 Rendering Equation	- 12 -
2.3 Materials	- 13 -
2.3.1 Diffuse Reflection	- 13 -
2.3.2 Specular Reflection	- 13 -
2.3.3 Refraction	- 14 -
2.3.4 Glossy Reflection	- 14 -
2.3.5 Phong Reflection Model	- 14 -
2.4 Monte Carlo Methods	- 15 -
2.5 Rendering Algorithms	- 17 -
2.5.1 Ray-Tracing	- 17 -
2.5.2 Path Tracing	- 18 -
2.5.3 Photon Mapping	- 19 -
2.5.4 Irradiance Caching	- 20 -
2.5.5 Radiosity	- 20 -
2.5.6 Instant Radiosity	- 22 -
 3 VISUALIZATION OF SPATIAL MODELS	 - 25 -
3.1 Systems for Modeling of Virtual Worlds	- 25 -
3.1.1 Autodesk 3D Studio Max	- 25 -
3.2 Systems for Visualization of Virtual Worlds	- 26 -
3.2.1 VRML	- 26 -
3.2.2 3D Studio Max to VRML Conversion	- 32 -
3.3 Systems for Image Synthesis of Virtual Worlds	- 32 -
3.3.1 PBRT	- 34 -
3.3.2 VRML to PBRT Conversion	- 39 -
 4 PREVIOUS WORK	 - 41 -
4.1.1 Data Model	- 41 -
4.1.2 Implemented Application	- 43 -
4.1.3 Previous Results	- 45 -
 5 ANALYSIS	 - 47 -
5.1 Application Architecture	- 47 -
5.2 Geometry Modifications	- 49 -
5.2.1 Data Model	- 49 -
5.2.2 Objects Description	- 52 -
5.2.3 Object Geometry Modification	- 53 -
5.3 Animations	- 54 -
5.3.1 Animation Nodes	- 55 -
5.3.2 Interpolation	- 56 -

5.3.3 Output Files Generation	- 59 -
5.4 Additional Extensions.....	- 60 -
5.4.1 Project Status	- 60 -
5.4.2 Console Version of VRMLtoPBRT Application	- 61 -
6 IMPLEMENTATION.....	- 63 -
6.1 Application Architecture.....	- 63 -
6.2 Data Model	- 64 -
6.2.1 Object Hierarchies.....	- 64 -
6.2.2 Adding New Lights	- 66 -
6.2.3 VRML Errors Corrections.....	- 67 -
6.3 Scene Representation.....	- 67 -
6.3.1 Data Structures	- 68 -
6.3.2 Object Recognition.....	- 70 -
6.4 Animations	- 71 -
6.4.1 Data Structures	- 71 -
6.4.2 Animation Specification.....	- 75 -
6.4.3 Interpolation	- 77 -
6.4.4 Animation File Processing	- 78 -
6.4.5 Animation Post-Processing	- 81 -
6.5 GUI Extensions	- 81 -
6.6 Additional Extensions.....	- 84 -
7 RESULTS	- 85 -
7.1 Geometry Modifications.....	- 85 -
7.2 Animations	- 86 -
7.2.1 Animation 1 – Static Camera in the Hall.....	- 87 -
7.2.2 Animation 2 – Dynamic Camera in the Hall	- 87 -
7.2.3 Animation 3 – Into the Atrium	- 87 -
7.2.4 Animation 4 – Entrance.....	- 88 -
7.2.5 Animation 5 – Roundel	- 88 -
8 CONCLUSION.....	- 95 -
9 REFERENCES	- 97 -
APPENDIX A – LIGHT TYPES CORRECTIONS	- 99 -
APPENDIX B – LIST OF MATERIALS	- 101 -
APPENDIX C – IMAGE GALLERY	- 105 -
APPENDIX D – PROJECT STATUS FILE FORMAT	- 111 -
APPENDIX E – USER MANUAL.....	- 119 -
E.1 Application GUI.....	- 119 -
E.1.1 Menu Options (A)	- 119 -
E.1.2 Information Panel (B).....	- 120 -
E.1.3 Data Conversion Options (C)	- 120 -
E.1.4 Objects Descriptions (D).....	- 125 -
E.1.5 Settings (E).....	- 128 -
E.2 User Guide.....	- 129 -

E.2.1	Scenario 1	- 129 -
E.2.2	Scenario 2	- 134 -
E.2.3	Scenario 3	- 137 -

APPENDIX F – DVD CONTENT	- 139 -
---------------------------------------	----------------

List of Figures

Fig. 1 The MPII Building, Saarbruecken, Germany	- 2 -
Fig. 2 Cornell Box dataset.....	- 6 -
Fig. 3 Aizu Atrium dataset.....	- 7 -
Fig. 4 Differential solid angle.....	- 8 -
Fig. 5 Computation of radiance.....	- 10 -
Fig. 6 Computation of the BRDF	- 11 -
Fig. 7 Pseudo-code of the MC solution to the rendering equation.....	- 17 -
Fig. 8 Pseudo-code of the ray-tracing algorithm.....	- 18 -
Fig. 9 Pseudo-code of the path tracing algorithm.....	- 19 -
Fig. 10 Pseudo-code of photon tracing phase of the photon mapping algorithm.....	- 20 -
Fig. 11 Pseudo-code of the radiosity method.....	- 22 -
Fig. 12 The structure of the VRML file.....	- 27 -
Fig. 13 Viewpoint node.....	- 28 -
Fig. 14 Shape node.....	- 28 -
Fig. 15 Appearance node.....	- 29 -
Fig. 16 Material node.....	- 29 -
Fig. 17 IndexedFaceSet node.....	- 30 -
Fig. 18 Transform node.....	- 30 -
Fig. 19 PointLight node.....	- 31 -
Fig. 20 SpotLight node.....	- 32 -
Fig. 21 PBRT main rendering loop.....	- 34 -
Fig. 22 Example of PBRT input file.....	- 35 -
Fig. 23 PBRT object instancing.....	- 36 -
Fig. 24 PBRT triangular mesh.....	- 37 -
Fig. 25 PBRT photo-goniometric light source.....	- 37 -
Fig. 26 PBRT uber material.....	- 38 -
Fig. 27 PBRT photon mapping surface integrator.....	- 38 -
Fig. 28 Model of the MPII building constructed by Josef Zajac.....	- 41 -
Fig. 29 Model of light and its gonio-photometric diagram.....	- 42 -
Fig. 30 Architecture of the implemented application.....	- 43 -
Fig. 31 Comparison of the results created by Jiří Drahoukoupil.....	- 44 -
Fig. 32 Application architecture design.....	- 48 -
Fig. 33 Object hierarchy in 3DS Max and after export into VRML.....	- 49 -
Fig. 34 Model of the door object and its corresponding 3DS Max model hierarchy.....	- 50 -
Fig. 35 Model of the window object and its corresponding 3DS Max model hierarchy.....	- 51 -
Fig. 36 Model of the shutter object and its corresponding 3DS Max model hierarchy.....	- 52 -
Fig. 37 Design of the animation workflow.....	- 55 -
Fig. 38 Comparison of the linear, cosine and cubic interpolation methods on a set of points.....	- 57 -
Fig. 39 Visualization of the KB Spline for different values of tension, bias and continuity.....	- 59 -
Fig. 40 Implementation of the data conversion application.....	- 63 -
Fig. 41 Examples of the modified MPII 3DS Max model.....	- 65 -
Fig. 42 The 6th floor in the MPII building without and with the added room lights.....	- 66 -
Fig. 43 Incorrectly transformed doors in VRML.....	- 67 -
Fig. 44 Animation specification.....	- 76 -
Fig. 45 Display of objects of interest.....	- 82 -
Fig. 46 Dialog for specifying the global animation properties.....	- 82 -
Fig. 47 Dialog for animating cameras.....	- 83 -
Fig. 48 Dialog for specifying the PBRT animation output.....	- 84 -
Fig. 49 Geometry modification of two types of door objects.....	- 85 -
Fig. 50 Geometry modification of the window object.....	- 86 -
Fig. 51 Geometry modification of the window shutter object.....	- 86 -
Fig. 52 Snapshots from the Animation 1- Static camera in the hall.....	- 89 -
Fig. 53 Snapshots from Animation 2 – Dynamic camera in the hall.....	- 90 -

Fig. 54 Snapshots from Animation 3 – Into the Atrium.	- 91 -
Fig. 55 Snapshots from Animation 4 – Entrance.	- 92 -
Fig. 56 Snapshots from Animation 5 – Roundel.	- 93 -
Fig. 57 Main panel of the VRMLtoPBRT application.	- 119 -
Fig. 58 Main options in the applicatiion GUI.	- 119 -
Fig. 59 Information panel of the GUI.	- 120 -
Fig. 60 Data conversion options.	- 121 -
Fig. 61 Rendering options dialog.	- 121 -
Fig. 62 Dialog for creating a PBRT output data for a static scene.	- 122 -
Fig. 63 Dialog for creating a VRML output data for an animated scene.	- 123 -
Fig. 64 Dialog for creating a PBRT output data for an animated scene.	- 124 -
Fig. 65 Object description panel.	- 125 -
Fig. 66 Camera modification dialog.	- 126 -
Fig. 67 Door modification dialog.	- 127 -
Fig. 68 Settings options.	- 128 -
Fig. 69 Global animation settings.	- 129 -

List of Tables

Table 1 Plug-in categories of the PBRT application. - 33 -

Table 2 Explanation of individual parametetrs of the object labeling format. - 53 -

Table 3 Effects of different parameter values on the KB spline. - 59 -

Table 4 Number of remodelled objects of interest. - 66 -

Table 5 List of prefix categories of the project file format. - 111 -

Table 6 Description of the project file format commands..... - 111 -

Table 7 List of output actions of the project file format..... - 115 -

1 Introduction

This chapter presents the introduction to the problem of predictive image synthesis. It briefly discusses the previous work on this project and states the motivation for this thesis. Finally, the structure of the work is laid out.

1.1 Predictive Image Synthesis

Image synthesis is one of the most expanded fields of modern computer graphics. Virtual models serve as input data for algorithms that visualize the given scene. In general, image synthesis can be divided into *non-photorealistic*, *photorealistic* and *predictive image synthesis*. Non-photorealistic image synthesis is not concerned with the realism of the produced images. On the other hand, the photorealistic image synthesis aims at predicting the appearance of the virtual scene in a real environment under the specified conditions with the highest degree of realism possible. It utilizes the knowledge about the physical phenomena of light, its transportation and interaction. This knowledge is transformed into algorithms for global image synthesis. The price for the high realism of the produced images is the typical high computational complexity.

However, the realism of the synthesized image is a relative matter. The assessment of this realism is the aim of the research field called the *predictive image synthesis*. Here, the input data model is constructed according to a real world object. The synthesized images can then be compared to the real world prototype and their realism and correctness can be assessed. This is in contrast to the commonly used photorealistic techniques, where the realism of the physical model cannot be guaranteed and the algorithms or the results are often tailored towards the specific needs and images are expected to be believable.

Clearly, predictive image synthesis has wide applications in areas such as design, architecture or film making. Unfortunately, suitable data models for predictive image synthesis are rather scarce. While some datasets are small and obsolete, others have low quality and are unrealistic. Even worse, most of the suitable datasets are owned by private companies and are rarely released for public use and research. Creation of a new publicly available dataset with sufficient complexity and accuracy would significantly contribute to the general research community.

This thesis is a part of an ongoing project that was initiated in year 2003. Its main aim is to create a dataset for predictive image synthesis based on the MPII (Max-Planck-Institut fuer Informatik) building in Saarbrücken, Germany.

1.2 Previous Work

The effort to model the MPII building was initialized by Dr. Vlastimil Havran and Josef Zajac in 2003. During year 2004 a major part of the 3D model of the MPII building was created. At that time, the major problems were the incorrect export from the used modeling software and the lack of accurate description of materials and luminaries.

Work on the project was renewed in 2007 by Dr. Havran's graduate student Jiří Drahokoupil, who pursued this project as his Master Thesis titled: "Predictive Image Synthesis from MPII Building Model" [1]. The major advances in the project can be summarized as:

- 1) Modeling and adding additional pieces of geometry (furniture, fences, handrails).
- 2) Measuring the luminaries for light sources and placing them into the scene.

- 3) Estimating the BRDF for materials.
- 4) Initial estimation of viewpoints.
- 5) Implementation of data conversion tool from VRML to PBRT file format.

Apart from adding additional parts of the geometry models, the main contributions to the realism of the model, were adding the BRDF for materials in the building and measuring the luminaries of the light sources. Due to the obvious difficulty with measuring the actual materials in the MPII building, the BRDF was approximated based on visual observations in the MPII building and available photographs of the building. Samples of actual light sources from the building were collected and measured at Czech Technical University in Prague.

In order to produce photorealistic visualizations of the created dataset, the PBRT (Physically Based Ray Tracing) software was chosen. One of its main advantages was that it is publically available application, which is widely used in the computer graphics research community. Further, its input format is relatively simple and thus suitable for data model conversion. As part of his thesis Jiří Drahekoupil implemented an application with graphical user interface (GUI) that enables the user to load scenes, attach materials, set the correct light sources and export the dataset into the PBRT format. The previous work is summarized in the MPI-I-2009-4-004 technical report [24].



Fig. 1 The MPII Building, Saarbruecken, Germany [1].

1.3 Current Work

The motivation for the presented work is the desire to further extend the current dataset and the implemented application. This extension is not achieved by adding additional details into the model. Instead, a parametric geometry model is to be created. In other words, the user should be able to easily modify the geometry of the movable parts of the model as they could be manipulated in real world. Such movable parts include doors, windows and

shutters along with their auxiliary manipulating elements such as door handles or window handles. Accordingly, the application with its GUI should be updated to allow the user for such modifications.

Furthermore, this parametric model can be used for producing animations of the whole dataset. This includes not only animation of the geometry modifications but also animations of the camera and the lights in the building.

Altogether, the presented thesis discusses the techniques and methods necessary for the implementation of the parametric model of the MPII building suitable for predictive image synthesis.

1.4 Thesis Structure

Chapter 2 gives a brief background overview of the problem of image synthesis. First, a detailed overview of the predictive image synthesis is presented. Next, the basic physical phenomena responsible for the transportation of light in a scene are discussed along with the description of materials and light sources. Finally, Monte Carlo methods are introduced as suitable means for photo-realistic image synthesis and several algorithms for global illuminations are described.

Chapter 3 introduces systems used for modeling, visualizing and image synthesis of virtual worlds. Specifically, the 3DS Max, VRML and the PBRT software tools are presented and their differences compared.

Chapter 4 reviews the previous work done in this project. The construction process and the current state of the data model are discussed as well as the functionalities of the implemented software application. Also, some results of the previous work are presented.

Chapter 5 presents analysis of the work considered in this thesis and proposes a design of the solution. The methods for implementing the geometry modifications of movable parts of the building are presented. Further, a methodology for specifying the animations of cameras and building geometry is proposed.

Chapter 6 follows by describing the implementation details of the presented work. Details of the modifications of the data model and object hierarchies are given along with the description of the implemented data structures. Further, the implementation of the geometry modifications and animations in the VRML and PBRT file formats are presented. Finally, the improvements to the GUI and additional new functionalities are summarized.

Chapter 7 shows the results of the data model conversion and the subsequent image synthesis. Examples of rendered geometry modifications are presented. Screenshots from the generated animation sequences are displayed.

Chapter 8 summarizes the presented work and compares the achieved results to the stated goals.

2 Image Synthesis

This chapter provides a background overview of computer based image synthesis. First, a general explanation of the predictive image synthesis is given. Next, the physical phenomena behind the light transportation and interaction are discussed. Finally, the Monte Carlo methods and several algorithms for photorealistic image synthesis are explained.

2.1 Predictive Image Synthesis

The basic motivation for the predictive image synthesis comes from the very principles of conducting research work in computer science. It is more than clear that the performance of the implemented algorithms is a subject to its input data. The performance of the algorithm depends on the size of the input dataset, its dimensionality, limited precision of data representation or its statistical properties (e.g. ordered sequences, uniform or clustered distributions). For those reasons, reports on algorithms and their performance also have to state the datasets used.

As stated in [3]: “The primary test of a scientific result is its reproducibility.” Hence, in order to reproduce the reported results, the used dataset has to be available as well. As discussed further, this constitutes a significant problem in the area of predictive image synthesis, since very few suitable datasets are publically available. Some suitable datasets are privately owned by companies and are not released to public. This has several drawbacks for conducting research. First, when used multiple times in publications, the private data enable the authors of the work to be traced. This has a negative impact on the unbiased blinded review process. Second, only the results of the work can be shown, but no data are provided to the research community. This means that the work can be only reproduced in theory but not in practice.

As it was pointed out in the work of Drago and Myszkowski [2], it is relatively easy to create great looking and artistic images, but it is substantially more difficult to match the appearance of real world environment as perceived by human observers. In order to achieve high degree of realism in the rendered images, a complex physically-based lighting simulation has to be used. Because analytical evaluation of the synthesized images with respect to human observers is generally impractical, the simulation result has to be experimentally checked against some reference data. Hence, the data model of the virtual scene should be constructed based on its real world counterpart and the conditions of the simulation process (including lighting and materials) should be modeled as precise as possible. The contribution of such predictive image synthesis is the option to validate the correctness of the implemented physically-based lighting algorithm.

Despite the clear benefits of this methodology, it has been seldom used in practice. The main reason is that lack of publically available test data, which would fulfill all the needs of predictive image synthesis. The question of “what constitutes a good test dataset” was answered in [2] as follows:

- The modeled scene should have a significant geometrical complexity. Only in such complex environment the behavior of the algorithm can be sufficiently tested. Further, such input data enable the analysis of the computational load and memory usage with respect to the complexity of the problem.
- Significant lighting complexity should be present. The scene should include different types of lights with different goniophotometric diagrams. These should illuminate parts of the scene with complex geometry. In addition, some areas of the scene should

be only illuminated by indirect lighting, while the direct lighting should dominate in other regions.

- Surfaces with various reflectance characteristics should be present. The scene should also contain materials with reflectance functions that are difficult to approximate with simplified diffuse or specular models.
- Most importantly, the modeled environment must exist in the real world. Only in this way, measurements of the lighting conditions and material properties can be performed and the perceptual validation of the obtained results is possible.

Several datasets have been previously developed. Probably the most well known dataset is the Cornell box [4], which is one of the most commonly used datasets in computer graphics. Its drawback is that it is rather small dataset with very simple geometry and lighting conditions. Comparison of the real world Cornell box and its rendered virtual model can be seen in Fig. 2. One rare example of large private dataset, which was released to public use, is the Boeing model, which is composed of 350 million triangles. However, the lack of reflectance and luminaries data and the fact that no reference photographs under specified lighting conditions were obtained, makes it unsuitable for predictive image synthesis.



Fig. 2 Cornell Box dataset. Real world model (a) and a rendered view of the virtual model (b) [4].

The validation of global illumination and rendering techniques was the motivation for the work of Drago and Myszkowski [2]. They created a detailed model of the atrium at the University of Aizu. They performed BRDF measurements of the six materials covering over 80% of the surface in the scene. The rest of the materials were approximated. The luminaries were measured and the maintenance factor of the lights in the scene was estimated to give the most accurate model. However, some problems with the Aizu dataset can be identified: 1) remote location of the building – it is located in Japan, 2) insufficient realism and complexity – only a single hall is modeled and no data are available behind the doors. Comparison of the photographed Aizu atrium and its rendered virtual model can be seen in Fig. 3.

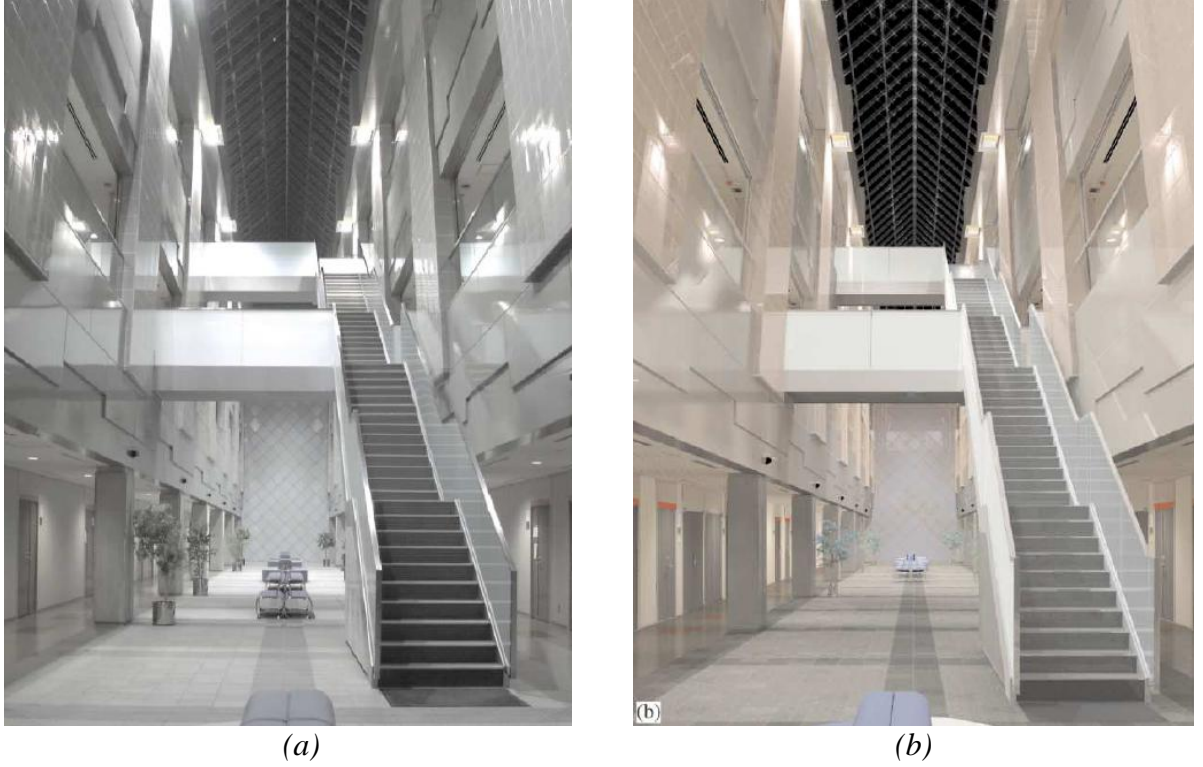


Fig. 3 Aizu Atrium dataset. Photograph (a) and rendered view of the virtual model (b) [2].

This thesis is a part of an ongoing project that attempts to resolve the above mentioned issues and difficulties of predictive image synthesis. Creation of a publically available dataset, which is specifically intended for predictive image synthesis, would be of a great use for the research community.

2.2 Rendering Equation

In order to setup the theoretical ground for the explanation of algorithms used for photo-realistic image synthesis, the general rendering equation is introduced and explained. First, some basic terms have to be defined.

Algorithms simulating the interactions of light with objects and optically active environment are fundamental for the whole field of computer graphics and for photorealistic image synthesis in particular. Despite attempting to simulate complex physical phenomena, some simplifying assumptions have to be imposed during the light simulation process [5]:

- Light radiates in a straight-forward manner.
- The speed of light is infinite. No dynamic effects are present in the scene.
- The light is unaffected by gravitation force, electromagnetic field and no relativistic effects take place.

Two distinct approaches for describing and quantifying light can be found in literature. *Radiometry* describes light as electromagnetic radiation regardless of its perception by human observer. *Photometry* describes the light with respect to the variable sensitivity of human eyesight. Hence, the algorithms for image synthesis typically use radiometric description of

light, which is less subjected to individual human observers. In the following text, only the basic radiometric quantities are explained. Their photometric counterparts are briefly mentioned.

2.2.1 Radiometry

This section discusses some of the important radiometric quantities.

Differential Solid Angle

First the concept of differential solid angle is explained due to its necessity for both radiometry and photometry.

A solid angle is determined by the area that a cone will cut off on the surface of a unit sphere with identical center. In general, a surface with area A projected on the surface of a sphere with radius r is seen from the center of the sphere under the solid angle ω that can be determined as:

$$\omega = \frac{A}{r^2} \quad [sr] \quad (2.1)$$

In a similar manner, the differential angle $d\omega$ of an element dA from a surface seen from distance l can be computed as:

$$d\omega = dA \frac{\cos \theta}{l^2} \quad [sr] \quad (2.2)$$

Here, θ is the angle between the axis of the differential solid angle $d\omega$ and the normal of the surface element dA as seen in Fig. 4.

Radiant Energy

The basic quantity of light is the radiant energy. It describes how many photons are present at certain location in space. It is commonly denoted by symbol Q and the unit is Joule. Its photometric counterpart is the luminous energy with the unit Talbot.

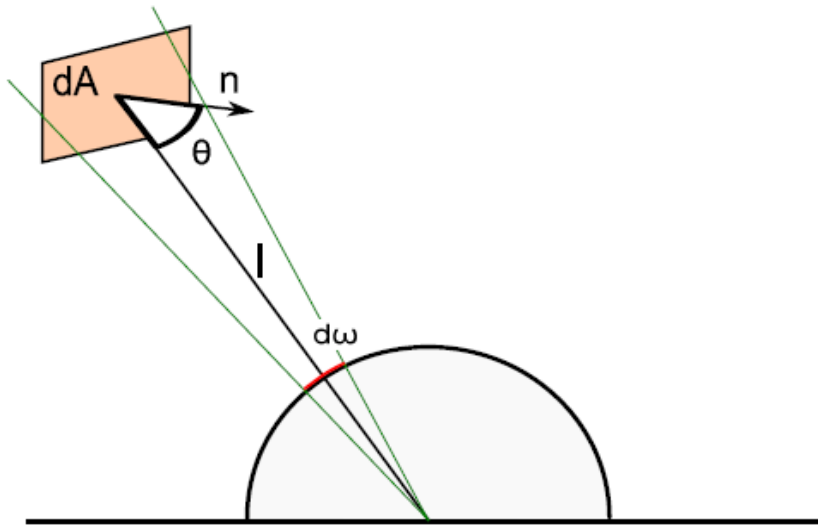


Fig. 4 Differential solid angle [1].

Radiant Flux

The radiant flux describes how fast the amount of photons changes in a certain location in space. It defines the energy transferred by the electromagnetic radiation in certain period of time. It is defined as:

$$\Phi = \frac{dQ}{dt} [W] \quad (2.3)$$

The unit is Watt. Its photometric counterpart is the luminous flux with the unit Lumen.

Irradiance

The irradiance defines the density of the flux. It characterizes the number of photons that hit a unit area in a specific place per unit of time. It can be computed as follows:

$$E(x) = \frac{d\Phi(x)}{dA} [W.m^{-2}] \quad (2.4)$$

Its photometric counterpart is the illuminance with the unit Lux.

Radiosity

The radiosity or the radiant excitance is similar to irradiance. It determines the amount of photons that are emitted from an element with the area of dA . It does not distinguish between self-emission and the reflected light. It is computed as:

$$B(x) = \frac{d\Phi(x)}{dA} [W.m^{-2}] \quad (2.5)$$

Its photometric counterpart is the luminosity with unit Lux.

Radiant Intensity

The radiant intensity describes the number of photons per differential solid angle that are emitted in the given direction. It is defined as follows:

$$I(\omega) = \frac{d\Phi(\omega)}{d\omega} [W.sr^{-1}] \quad (2.6)$$

The photometric counterpart of radiant intensity is the luminous intensity with unit Candela. The Candela unit is the base unit of SI.

Radiance

The radiance is the most fundamental quantity for photorealistic image synthesis. It describes the number of photons radiating in a certain direction per unit time through a surface with differential area dA . It can be computed as:

$$L = \frac{d^2\Phi(\omega)}{d\vec{\omega}.d\vec{A}} [W.sr^{-1}] \quad (2.7)$$

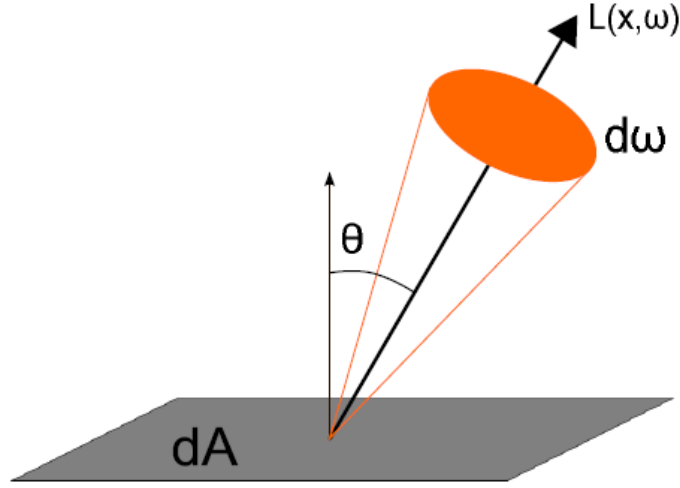


Fig. 5 Computation of radiance [1].

This can be further modified by the following substitution:

$$d\vec{\omega}.d\vec{A} = d\omega.dA.\cos\theta \quad (2.8)$$

Here, θ is the angle between the axis of the differential solid angle $d\omega$ and the normal of the surface element dA as shown in Fig. 5. This yields the typically used formula for the radiance:

$$L = \frac{d^2\Phi(\omega)}{d\omega.dA.\cos\theta} \quad [W.sr^{-1}] \quad (2.9)$$

The radiance is fundamentally important because all other radiometric quantities can be derived from it. Further, the radiance determines the color and the intensity of the light when looking at certain object. Two features of radiance that are particularly important for image synthesis are:

- The radiance is constant along the path of the ray. The only exception is the optically active environment (e.g. fog, smoke or atmosphere), where the photons can change their trajectories due to interaction with particles of the medium.
- The response of human eye or a camera is directly proportional to the amount of received radiance.

The photometric counterpart of radiance is the luminance with unit Candela per square meter.

2.2.2 Bidirectional Reflectance Distribution Function

The color of objects, as perceived by human eye, is given by the spectral composition of the light that is lighting the object as well as by its surface properties. The surface structure and properties define how particular wavelength get affected by the interaction and in what direction the light will be emitted back to the environment. In computer graphics, this interaction is described by the Bidirectional Reflectance Distribution Function (BRDF). For the definition of the BRDF several assumptions are necessary:

- The light is reflected immediately without any delays. This is in agreement with the infinite speed of light requirement stated earlier.
- A photon of certain wavelength will be emitted at the same wavelength.
- A photon interacting with certain spatial location x will also be emitted from this exact location. Hence, BRDF does not consider sub-surface scattering of light.

Under those assumptions, the BRDF (denoted by symbol f) at certain point x can be defined as the ratio of the reflected radiance L_r into a certain direction ω_r to the input radiance L_i incoming from a certain direction ω_i :

$$f(x, \vec{\omega}_r, \vec{\omega}_i) = \frac{dL_r(x, \vec{\omega}_r)}{dL_i(x, \vec{\omega}_i)(\vec{\omega}_i \cdot \vec{n})d\vec{\omega}_i} \quad (2.10)$$

Fig. 6 visualizes the computation of the BRDF function.

Several properties of the BRDF function are fundamental for the implementation of the algorithms for image synthesis [5]:

- *Helmholtz's principle of reciprocity* declares that the value of BRDF remains constant at the given point, when we swap the incoming and the reflecting directions:

$$f(x, \vec{\omega}_r, \vec{\omega}_i) = f(x, \vec{\omega}_i, \vec{\omega}_r) \quad (2.11)$$

This principle is crucial for computer graphics, because it allows for bidirectional computation of light in the scene.

- *Positivity* of the BRDF defines that the values of the function are never negative.

$$f(x, \vec{\omega}_r, \vec{\omega}_i) \geq 0 \quad (2.12)$$

- The *law of energy conservation* defines that the energy cannot originate or cease

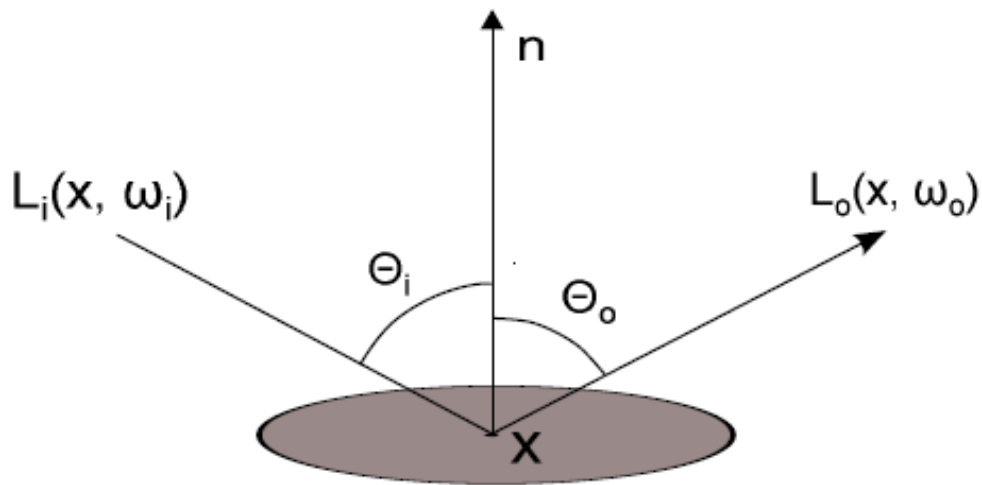


Fig. 6 Computation of the BRDF [1].

somewhere. It can only be transformed from one kind to another one. This can be expressed in the following manner:

$$\int_{\Omega} f(x, \vec{\omega}_r, \vec{\omega}_i) \cos \theta_i d\vec{\omega}_i < 1, \forall \vec{\omega}_i \quad (2.13)$$

- *Linearity* declares that the value for a specific incoming angle $\vec{\omega}_i$ is independent from the values of BRDF for other incoming angles.

Several representations of the BRDF can be found. Typically, BRDF can be expressed as a relatively simple empirically derived function. In more complex cases, BRDF can be stored as a multi-dimensional look-up table. The actual value is the obtained by interpolation.

2.2.3 Rendering Equation

The rendering equation constitutes a mathematical formulation of propagation of light in the scene. The solution of this equation defines the outgoing radiance for every point and direction in the scene. This radiance can then be transformed to the color of a given pixel in the camera. The basis for the rendering equation is the law of energy conservation. Since the rendering equation is a complex integral, its analytical solution cannot be obtained in the general case and an approximate solution has to be found.

First, the local interaction between the light and the surface can be described by the local lighting model as:

$$L_r(x, \vec{\omega}_r) = \int_{\Omega} f(x, \vec{\omega}_r, \vec{\omega}_i) L_i(x, \vec{\omega}_i) \cos \theta d\vec{\omega}_i \quad (2.14)$$

Here, function $f(x, \vec{\omega}_r, \vec{\omega}_i)$ is the BRDF function. This local lighting model computes the outgoing radiance L_r in the direction $\vec{\omega}_r$ for incoming radiance L_i from all directions $\vec{\omega}_i$.

Next, the total radiance leaving point x in the direction $\vec{\omega}_r$ is given by the sum of the reflected light and the self-emission at point x :

$$L_o(x, \vec{\omega}_o) = L_e(x, \vec{\omega}_o) + \int_{\Omega} f(x, \vec{\omega}_o, \vec{\omega}_i) L_o(h(x, -\vec{\omega}_i), \vec{\omega}_i) \cos \theta d\vec{\omega}_i \quad (2.15)$$

Here, function $h(x, -\vec{\omega}_i)$ is a function that returns a point in the scene, which is visible from point x in the direction $-\vec{\omega}_i$. Hence, the integral on the right side computes the sum of the radiance contributions of all visible surfaces in the scene around point x . These radiance contributions are scaled by the BRDF function and by the projection angle $\cos \theta$.

This rendering equation belongs to the group of so called second degree Fredholm's equations. Its major complication is that the unknown outgoing radiance is present on both left and right side of the equation. Since it describes the transportation of energy between different points in the whole scene, it constitutes a global lighting model.

For the implementation of the image synthesis algorithms, the presented form of the rendering equation with the integral over the hemisphere around point x , might not be suitable. The equation can be rewritten using an integral over all the contributing surfaces:

$$L_o(x, \vec{\omega}_o) = L_e(x, \vec{\omega}_o) + \int_S L_o(y, y \rightarrow x) f(x, y \rightarrow x, \vec{\omega}_o) G(x, y) V(x, y) dA \quad (2.16)$$

Several new terms were introduced in equation (2.16). Term $y \rightarrow x$ denotes the direction from point y to point x . It can be compute as:

$$y \rightarrow x = \frac{x - y}{\|x - y\|} \quad (2.17)$$

The geometrical term $G(x, y)$ reflects the projection of the outgoing radiance at point y and incoming to point x . It can be expressed as:

$$G(x, y) = \frac{\cos \theta_x \cos \theta_y}{\|x - y\|^2} \quad (2.18)$$

Finally, the visibility term $V(x, y)$ yields number 1, when point x is visible from point y and number 0 otherwise.

2.3 Materials

This section discusses several fundamental processes that occur when the light interacts with the surface of certain material. The properties of the microstructure of the surface influence the composition of the reflected light and the direction in which it is reflected. As described earlier the interaction between the light and the surface of an object can be described by the BRDF. Further, the Phong reflection model is explained [5].

2.3.1 Diffuse Reflection

The diffuse reflection is a special case of the general reflection model. Perfectly diffuse surface reflects the light evenly to all outgoing directions. The BRDF is therefore constant for any combination of the incoming direction $\vec{\omega}_i$ and the outgoing direction $\vec{\omega}_r$:

$$f_d(x, \vec{\omega}_r, \vec{\omega}_i) = f_d(x) = \text{const.} \quad (2.19)$$

Given this fact, the local lighting model stated in (2.14) can be rewritten as:

$$L_r(x, \vec{\omega}_r) = f_d(x) \cdot E_i(x) \quad (2.20)$$

Here, $E_i(x)$ is the irradiance at point x .

2.3.2 Specular Reflection

Another important phenomenon is the specular reflection. Specular reflection is typical for mirrors, polished metals or water surface. Perfect specular reflection appears when the incoming radiance is reflected into a single outgoing direction. This direction can be easily determined as follows:

$$\vec{\omega}_r = 2(\vec{\omega}_i \cdot \vec{n})\vec{n} - \vec{\omega}_i \quad (2.21)$$

Here, vector \vec{n} denotes the normal to the surface at the incidence point x .

2.3.3 Refraction

Refraction of light can be observed when the light passes between materials with different refractive indices. The refractive indices express the optical density of the environment and thus influence the speed of light. The incoming beam of light is decomposed into two parts: the reflected and the refracted light. The direction of the refracted light can be determined from the Snell's law, which states:

$$\frac{\sin \theta_i}{\sin \theta_r} = \frac{\eta_r}{\eta_i} \quad (2.22)$$

Here, symbols η_i and η_r denote the refractive indices of the first and the second medium, respectively. In the implementation of particular algorithms for image synthesis, it is necessary to find the direction of the refracted light $\vec{\omega}_r$ given the direction of the incoming light $\vec{\omega}_i$ and knowing the refractive indices of both media. This can be computed from the following equation:

$$\vec{\omega}_r = \frac{\eta_i}{\eta_r} \vec{\omega}_i - [\vec{\omega}_i - (\vec{\omega}_i \cdot \vec{n}) \vec{n}] - \vec{n} \sqrt{1 - \left(\frac{\eta_i}{\eta_r} \right)^2 [1 - (\vec{\omega}_i \cdot \vec{n})^2]} \quad (2.23)$$

2.3.4 Glossy Reflection

Substantially more complex phenomenon is the glossy reflection. It is a composition of multiple complex interactions, which interact with each other. This model assumes that the structure of the micro surface is composed of micro-facets, which enable the light to travel under the surface. One of the simplified formulas for the BRDF of the glossy reflection can be written as follows:

$$f_r(x, \vec{\omega}_r, \vec{\omega}_i) = \frac{D \cdot G \cdot F}{4 \cos \theta_r \cos \theta_i} \quad (2.24)$$

Here, D describes the distribution of the micro-facets, G is a geometrical element simulating the self-shadowing of the material, and F denotes the Fresnel coefficient of reflection.

2.3.5 Phong Reflection Model

Probably the most common empirically derived model for computing the reflected light is the Phong reflection mode, introduced by Bui-Tuong Phong in 1977 [6]. Phong reflection model decomposes the reflected light into three components: specular, diffuse and ambient light. These components are then added together to create the final reflection model.

The specular light I_s simulates rather the glossy reflection than the perfectly specular reflection. It is calculated as follows:

$$I_s = I_L r_s (\vec{v} \cdot \vec{r})^h \quad (2.25)$$

Here, I_L symbolizes the incoming light, coefficient r_s weights the specular reflection with respect to the other types of reflected light, direction \vec{v} is the normalized direction towards the observer, h is a coefficient determining the sharpness of the specular reflection, and direction \vec{r} is the direction of the ideal specular reflection, which was defined in (2.21).

The diffuse component of the light I_d can be understood as the reflection responsible for the color of the object. It can be computed as:

$$I_d = I_L r_d (\vec{l} \cdot \vec{n}) \quad (2.26)$$

Here, r_d is the coefficient of the diffuse reflection, \vec{l} is the direction of the incoming light and \vec{n} is the surface normal at the given point of incidence.

Phong reflection model approximates the indirect illumination, which is caused by multiple reflections of the incoming light from other objects, by the ambient light I_a . It is defined in the following form:

$$I_a = I_A r_a \quad (2.27)$$

Here, r_a is the coefficient of the ambient reflection and I_A defines the amount of the ambient light presented in the scene.

2.4 Monte Carlo Methods

The Monte Carlo (MC) methods are of a great importance to the photorealistic image synthesis due to their use for evaluation of the global rendering equation. These methods are closely related to statistical sampling, which has been used since the 18th century. However, their true birth can be seen in the construction of the first electronic computer – the ENIAC – at the University of Pennsylvania, Philadelphia in 1945. This first computer was enormous in size and consisted of some 18.000 double triode vacuum tubes and about 500.000 solder joints. Quite interestingly the computer was constructed from rejected army-navy parts, meaning that its construction was possible already before the second world war started for the USA in 1941 [7]. But more importantly this first electronic computer had enough computational power to convert the statistical sampling method into a practical and useful tool.

In 1947, having realized the capabilities of the electronic computer, Stanislaw Ulam and John von Neumann proposed a solution to the problem of calculating the neutron diffusion and multiplication in an assembly of fissile material using statistical sampling. Later on this method was named by Nicholas Metropolis the Monte Carlo method, inspired by Stanislaw Ulam's uncle, who always wanted to go and gamble in Monte Carlo [7].

The most general applications of MC methods is the enumeration of a finite integral I of function $f(x)$ on an interval from a to b . This can be written as:

$$I = \int_a^b f(x) dx \quad (2.28)$$

The basic idea behind the MC integration is the stochastic sampling of the given interval and applying function $f(x)$ to the obtained samples. The mean value of these samples multiplied by the length of the interval then converges towards the value of integral I . Taking N samples x_i yields the result:

$$I_m = (b-a) \frac{1}{N} \sum_{i=1}^N f(x_i) \quad (2.29)$$

$$\lim_{n \rightarrow \infty} I_m = I \quad (2.30)$$

The variance of the integrand f can be expressed as:

$$\text{var}(f) = \frac{1}{N} \sum_{i=1}^N (f(x_i) - E(f))^2 \quad (2.31)$$

Using the central limit theorem the variance of the MC integration can be formulated as:

$$\text{var}(f) = (b-a)^2 \frac{\text{var}(f)}{N} \quad (2.32)$$

The error of the method corresponds to the standard deviation. Hence:

$$\text{Error}(I_m) = \sigma_m = (b-a) \frac{\sigma_f}{\sqrt{N}} \quad (2.33)$$

The derivation above showed that the MC methods yield relatively slow convergence. Namely, they provide $1/\sqrt{N}$ convergence. This means that using four times as many samples to compute the integral will reduce the error only in half.

Solution to this problem are the quasi-Monte Carlo methods. Here, the random number generator is substituted by a quasi-random number generator. Such generators are guided by an underlying function, which reduces randomness in the generated samples. Many types of quasi-Monte Carlo generators exist, for instance van der Corput, Halton, Hammersley, Sobol or Niederreiter, to name a few. Using the quasi-Monte Carlo changes the convergence of the computation can be expressed as:

$$O\left(\frac{\log N}{N}\right)^s \quad (2.34)$$

Here, s is the dimensionality of the problem domain. Hence, for lower dimensional problems (such as computer graphics), those method will experience faster convergence. The downside is that apparent visual artifacts can appear in computed images.

Examples of the many applications of Monte Carlo methods in photorealistic image synthesis are solving the rendering equation, rendering area lights, antialiasing, or computing the form factors for radiosity methods.

The rendering equation (2.16) contains integration over all incoming radiances at the incident point. The MC methods constitute a solution to this integral. The incoming directions can be sampled using a suitable probability distribution (BRDF) and an approximate solution can be obtained. The pseudo-code of the MC solution to the rendering equation at point x using N samples is shown in Fig. 7.

Clearly, a recursion can be seen in step 1.4, where in order to return the desired outgoing radiance another rendering equation has to be solved at the intersection point. This problem is solved in different ways by various algorithms for photo-realistic image synthesis, some of which are described in the following section.

1. Repeat N times
 - 1.1 Choose a random direction on the hemisphere around point x using the specific probability function.
 - 1.2 Shoot a ray in the given direction.
 - 1.3 If the ray does not intersect any geometry, return the outgoing color of the background.
 - 1.4 If the ray intersects with some geometry, return the outgoing radiance from the intersection point.
 - 1.5 Accumulate the returned radiance.
2. Compute an average value from the N samples.
3. Multiply the result by the reflection term $f(x, \vec{\omega}_r, \vec{\omega}_i) \cos \theta$.

Fig. 7 Pseudo-code of the MC solution to the rendering equation.

2.5 Rendering Algorithms

Algorithms for photo-realistic image synthesis were developed for computing an approximate solution to the rendering equation (2.16). Solving this equation in a complex scene, where lighting is a result of multiple interaction of the light with many objects, constitutes a computationally very expensive task. The algorithms can be roughly divided based on the directionality of the computation process.

First group of algorithms computes the solution starting from the observer. The color of each pixel on the screen is determined by shooting a ray through the pixel and computing where it came from in the scene. Such methods are taking advantage of the Helmholtz's principle of reciprocity, when the BRDF function can be used despite the fact that we swap the incoming and the outgoing directions. The advantage of those methods is that only rays of lights coming directly into the camera are considered.

Second group of algorithms is based on the computation of the scene illumination starting from the sources of lights. Such methods follow all possible trajectories of the emitted light and then display the result on the screen. Clearly, the number of simulated photons emitted from each light source has to be immense in order for the scene to be illuminated correctly and with an acceptable level of noise.

Third group of algorithms are methods combining the previous two categories. They use bidirectional simulation of light trajectories in order to combine the best from both groups. The underlying assumption is that some phenomena are more suitable for computation starting from the observer, while others are easier to compute when starting from the light source. For instance, the visualization of caustics is only possible when starting from the light source.

The main task of the algorithm for photo-realistic image synthesis is to create a simulation of scene illumination with the highest level of physical realism possible. Among many physical phenomena that are to be simulated are: glossy and mirror-like reflections, translucency and transparency, soft shadows, caustics, diffuse inter-reflection, surface scattering or refraction and dispersion.

2.5.1 Ray-Tracing

The original ray-tracing algorithm was introduced in 1980 in the work of Whitted [17]. It constitutes one of the most fundamental techniques for computing the global illumination in a virtual scene. The algorithm starts by tracing rays coming from the observer through every

1. Cast a ray through the camera.
2. Find the closest intersection of ray and the scene geometry.
3. If no intersection is found return the background color.
4. Send shadow rays towards all light sources.
5. Determine the contribution of all visible light sources to the illumination at the intersection point.
6. If the refraction coefficient is greater than zero, trace the secondary refracted ray.
7. If the specular reflection coefficient is greater than zero, trace the secondary reflected ray.
8. Accumulate the contribution of all primary and secondary rays and return the computed color.

Fig. 8 Pseudo-code of the ray-tracing algorithm.

pixel of the image being synthesized. The rays, called *primary rays*, enter the scene and the first intersection with the geometry of the scene is found. If no intersection is encountered, the color of the background is returned. From the intersection, the *shadow rays* are cast towards every light source in the scene in order to determine if the light source is visible and therefore illuminates the point of intersection. The contributions of all visible light sources are accumulated. Furthermore, two *secondary rays* are cast to investigate the refracted and the reflected light. The direction of the secondary rays can be computed according to the reflection law and the Snell's law of refraction as explained in Sections 2.3.2 and 2.3.3. Hence, the ray-tracing algorithm approximates the integral form of the rendering equation by summation of the three components – direct lighting, refracted lighting, and reflected lighting. The assumption is that these are the dominant contributions to the integral. A pseudo-code of the ray-tracing method is given in Fig. 8.

When the new refraction and reflection rays are generated, the algorithm becomes recursive. A certain termination criterion has to be established to determine the maximal depth of recursion.

By its nature, the classic ray-tracing method can compute the direct as well as the indirect illumination. However, only the refraction and reflection of light is included in the indirect lighting. Furthermore, the method computes shadows. However, those are only hard shadows unlike in real world. Ray-tracing is known for its high computational complexity. These issues can be alleviated by using suitable data structures for representing the geometry of the scene. This leads to more efficient computation of the ray-geometry intersections.

2.5.2 Path Tracing

The path tracing method is a generalization of the Whitted's ray tracing algorithm. It was first introduced in 1986 by Kajiya [17]. It employs the Monte Carlo methods to compute the complete solution of the rendering equation (2.16). It is also sometimes referred to as the Monte Carlo path tracing.

Similarly to the ray-tracing algorithm the rays are traced starting from the observer. The path tracing algorithm also considers the contribution of the indirect illumination from diffuse surfaces. It can also work with area light sources (handles soft shadows), caustics and color bleeding.

The algorithm casts a large amount of rays through a single pixel of the camera. At each intersection point the local illumination model is evaluated. This includes shooting the shadow rays towards all light sources in the scene. Next, the BRDF of the material at the incidence point is randomly sampled to generate the directions of the secondary rays. Not

1. If the recursion depth exceeds the maximum depth, return the color of the ray
2. Find the closest intersection of the ray and the scene geometry
3. If no intersection is found return the background color
4. Send shadow rays towards all light sources.
5. Determine the contribution of all visible light sources to the illumination at the intersection point.
6. Sample the BRDF at the incidence point to determine the direction of the reflected ray.
7. If the refraction coefficient is greater than zero, trace the secondary refracted ray.
8. If the specular reflection coefficient is greater than zero, trace the secondary reflected ray.
9. Accumulate the contribution of all primary and secondary rays and return the computed color.

Fig. 9 Pseudo-code of the path tracing algorithm.

only the two rays for refraction and specular reflection are considered here, but the whole hemisphere is sampled according to the known BRDF. This constitutes the fundamental difference when compared to the classical ray-tracing algorithm.

The secondary rays are then traced in a recursive manner. This brings up a termination and computational complexity issues. If several rays are sampled at the incidence point from the BRDF, the number of rays increases exponentially. Therefore only a single secondary ray is casted. Instead, many primary rays are shot through every pixel of the camera and traced further as they proceed through the scene. By averaging their contributions, the estimate of the integral rendering equation is obtained. Because of using the Monte Carlo methods, different ray samples will be obtained each time the method is executed. When using insufficient number of samples, the method will result in high amount of noise. There is a clear tradeoff between the quality of the synthesized images and the computational complexity of the algorithm. The recursion depth of the algorithm has to be limited by a certain maximum depth value.

A pseudo-code of the path tracing algorithm can be seen in Fig. 9. As denoted earlier, the major difference to the ray tracing algorithm is in step 6, where the BRDF is sampled to obtain the direction of the secondary rays.

2.5.3 Photon Mapping

Photon mapping is an example of bidirectional algorithms. It is composed of two main phases. In the first phase – photon tracing – photons are traced from each light source and the photon maps are created. In the second phase – distributed ray tracing – rays are cast from the observer through the camera and the photon maps are utilized for computing the global illumination.

The pseudo-code of the photon tracing phase is presented in Fig. 10. The algorithm starts by randomly selecting one of the light sources with a probability proportional to its intensity. Then the starting position and the initial direction of the photon are chosen by sampling the emission distribution function of that particular light source. This results in all photons having approximately the same flux. The tracing of individual photons is identical to the path tracing mechanism because of the BRDF being reciprocal. Every time photons interact with diffuse (but not specular) surface they are stored in the photon map. During the photon generation phase, the photon map is implemented as a linear list of photons. After all photons are generated, the map is transformed into a kD-tree, which is needed for a better

1. Choose light source with probability proportional to its intensity.
2. Choose the starting position and the direction of the emitted photon based on the emission distribution function of the light.
3. Trace the ray.
4. Decide if to create a photon on a diffuse surface.
5. Compute the termination based on the albedo of surface hit.
6. Repeat until there are enough photons in the scene.

Fig. 10 Pseudo-code of photon tracing phase of the photon mapping algorithm.

performance. Each photon record in the map stores its position, incoming direction and the energy of the photon. Typically, two photon maps are constructed, the global photon map and the caustic photon map. The global photon map contains indirect diffuse lighting. The caustic photon map contains only focused indirect lighting, which previously interacted with a specular or transparent material – such as glass.

The second rendering phase is very similar to the path tracing algorithm. The recursion on diffuse and glossy surfaces is substituted for a radiance estimate based on the photon map. The direct lighting is computed in a classical manner, by shooting shadow rays towards all light sources. Ideal specular and refraction rays are also computed by shooting two rays in the know directions. However, the caustics and indirect illumination is computed using the photon maps. The final gathering is performed at each incidence point. From each point hundreds gathering rays are casted and the contributions of the nearest photons in the map are accumulated. Alternatively, the photon map can be used directly, which results in decreased quality of the final images. However, the computational burden is significantly reduced.

2.5.4 Irradiance Caching

The irradiance caching is an additional mechanism for speeding-up the computation of the diffuse component of indirect lighting. During indirect lighting calculation, many rays have to be sampled from the hemisphere over the point of incidence. However, due to the spatial coherence, the diffuse illumination is changing very slowly over the surfaces in the scene. This fact constitutes the major motivation for the irradiance caching algorithm. The hemisphere is sampled only at certain points and the result is stored into a cache. Around the stored points the diffuse lighting is interpolated from the cache.

The indirect irradiance can be computed either by using recursive ray-tracing or by exploiting the photon maps. When interpolating from cache, it is important to decide, which irradiance records can be still used and which ones are too far away. This is determined by the geometry of the scene. The irradiance is changing more rapidly around complex geometry, whereas on flat surface it is changing at lower pace. Hence, the radius for reusing particular irradiance records can be adaptively computed by averaging the length of rays casted from particular point. This will automatically result in higher density of irradiance cache records around complex geometry and lower density on open surfaces. The irradiance records are typically stored in an octree data structure.

The quality of the interpolation can be substantially improved by estimating the gradient of the irradiance on the surface. These gradient estimates are stored together with the irradiance records in cache.

2.5.5 Radiosity

The radiosity method constitutes one of the first attempts towards implementing a physically based photo-realistic image synthesis [19]. This method was inspired by the

advances in the heat transfer research field. The main assumptions made by the radiosity method are the followings:

- The amount of light energy transferred between objects in the scene is conserved. The scene is closed.
- All objects have perfectly diffuse surfaces.
- The objects are represented with boundary representation. This representation is composed of set of elements such as triangles or quadrilaterals.

The physical meaning of radiosity is that it is the rate at which the energy radiates from certain surface per unit area. The radiosity method starts by determining all light interactions present in the scene. These light interactions are determined in a view-independent way. Next, the scene can be rendered from any view by simply evaluating what part of the geometry is visible and applying some interpolation technique to achieve accurate shading.

Since the radiosity method only considers purely diffuse surfaces, the BRDF will be constant regardless of the incoming and the outgoing angle $\vec{\omega}_i$ and $\vec{\omega}_o$, respectively. The surface form of the rendering equation (2.16) can then be reformulated as follows:

$$L_o(x, \vec{\omega}_o) = L_e(x, \vec{\omega}_o) + \frac{\rho(x)}{\pi} \int_s L_o(y, y \rightarrow x) G(x, y) V(x, y) dA \quad (2.35)$$

Here, $\rho(x)$ is the coefficient of diffuse reflection at point x . The outgoing radiance can be expressed as the radiosity independent of the outgoing angle $\vec{\omega}_o$ as follows:

$$B(x) = B_e(x) + \rho(x) \int_s B(y) \frac{G(x, y) V(x, y)}{\pi} dA \quad (2.36)$$

The radiosity at point x is a sum of the self-emission and the radiosity coming from every visible element in the scene. The geometrical term inside the integral can be substituted as follows:

$$G' = \frac{G(x, y) V(x, y)}{\pi} \quad (2.37)$$

One of the main assumptions of the radiosity method is that the surfaces of all objects in the scene are represented with finite elements and the radiosity of each element is constant. By using the introduced substitution, (2.36) can be rewritten as:

$$B(x) = B_e(x) + \rho(x) \sum_{j=1}^N B_j \int_{A_j} G'(x, y) dA_j \quad (2.38)$$

The average (constant) radiosity of element i can be computed as the mean value of the radiosity over the surface of the element:

$$B_i = \frac{1}{A_i} \int_{A_i} B(x) dA_i = B_{e,i} + \rho_i \sum_{j=1}^N B_j \frac{1}{A_i} \iint_{A_i A_j} G'(x, y) dA_j \quad (2.39)$$

Next, the form factor can be extracted from the above equation. It determines how much of the light irradiated from element j is received directly by element i :

$$F_{i,j} = \frac{1}{A_i} \iint_{A_i A_j} G'(x, y) dA_j \quad (2.40)$$

Finally, the main equation of radiosity for each element in the scene has been derived:

$$B_i = B_{e,i} + \rho_i \sum_{j=1}^N B_j F_{i,j} \quad (2.41)$$

By considering all N elements, a linear system with N equations and N variables is formed:

$$\begin{bmatrix} 1 - \rho_1 F_{1 \rightarrow 1} & -\rho_1 F_{1 \rightarrow 2} & \cdots & -\rho_1 F_{1 \rightarrow n} \\ -\rho_2 F_{2 \rightarrow 1} & 1 - \rho_2 F_{2 \rightarrow 2} & \cdots & -\rho_2 F_{2 \rightarrow n} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_n F_{n \rightarrow 1} & -\rho_n F_{n \rightarrow 2} & \cdots & 1 - \rho_n F_{n \rightarrow n} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} B_{e,1} \\ B_{e,2} \\ \vdots \\ B_{e,n} \end{bmatrix} \quad (2.42)$$

This linear system can be solved by one of the available methods, for instance the Gauss elimination, or the Jakobi or Gauss-Seidel iteration methods.

The main advantage of the radiosity method is that it provides a view-independent solution and it allows for a very fast computation in simple scene. On the other hand, the disadvantages are that only diffuse surfaces are considered and that geometrical errors can occur due to the used final elements.

The pseudo-code of the radiosity method can be found in Fig. 11.

1. Divide the scene into elements
2. Compute the form factors F_{ij} .
3. Solve the system of linear equations. Compute the constant radiosity for each element in the scene
4. Set up the viewpoint and render the image
5. Set up the viewpoint and render the image
6. Set up the viewpoint and render the image
7. ...

Fig. 11 Pseudo-code of the radiosity method.

2.5.6 Instant Radiosity

The instant radiosity method was first introduced in the work of Keller [20]. Unlike the above mentioned radiosity method, the instant radiosity omits the computationally expensive scene partitioning into finite elements and the calculation of the form factors. This

is achieved by creating additional virtual point lights in the scene. The algorithm then uses these virtual lights to approximate the indirect illumination in the scene. In the first phase of the computation, photons are casted from each real light source. The photons are traced and a virtual light source is created after each bounce of the surface. In the second rendering phase, each virtual light source contributes to the indirect illumination of the scene.

The tracing of photons uses quasi Monte Carlo methods for choosing the light source, the starting point and the initial direction of the photon. Upon interacting with the surface, the photon can be either absorbed or it is reflected in a new direction.

.

3 Visualization of Spatial Models

This chapter discusses the means for modeling, visualization and image synthesis of virtual models. General principles as well as specific software tools are described.

3.1 *Systems for Modeling of Virtual Worlds*

As stated in [11], 3D modeling in computer graphics is the process of developing a mathematical representation of any three-dimensional object via specialized software. The models can be created automatically (e.g. procedural modeling) or manually.

In general, the modeled objects can be represented in two ways [5]. First, models can be represented as a solid volume. While being more realistic, these models are substantially more difficult to build and to work with. Second, models can be represented by a boundary representation. In this case, only the surface of those objects is modeled, typically by some kind of a mesh representation. Because the appearance of virtual objects depends entirely on their surface, the boundary representation is typically used in computer graphics.

Several ways of object modeling are also possible [11]. Probably the most common is polygonal modeling. In this case, points in 3D space are connected by edges that create polygons. The advantage of this modeling is its speed because modern computer systems have been optimized for displaying polygonal models. The disadvantage is that polygons are only an approximation to the real geometry of the object and thus geometry errors can be visible if the level of detail is not appropriately adjusted. An example of a software tool using this type of object modeling is the 3D Studio Max [12]. Objects can also be created by NURBS modeling. These models have surfaces that are described by spline curves with specified weight points. Software tools Maya or Rhino 3D implement the NURBS modeling. Similar modeling techniques are splines and patches. Another modeling method uses geometrical primitives. Here, complex models are created as a composition of basic geometrical primitives such as spheres, cylinders, cones or cubes. While, these objects can be precisely mathematically described, they are not capable of representing complex irregular shapes. An additional type of modeling is the sculpt modeling, which resembles the process of creating a sculpture in a real world.

Furthermore, the modeling software should provide a convenient tool for management and arrangement of the virtual scene. This includes placing virtual objects into the scene, creating lights or cameras and defining animations. The modeling software should also implement a rendering engine for visualization of the modeled scene and a conversion module for exporting the scene into other file formats.

Next, the 3D Studio Max will be described in more detail. The 3DS Max was previously chosen as a modeling tool for the static version of the MPI building model.

3.1.1 Autodesk 3D Studio Max

The Autodesk 3D Studio Max (3DS Max) is a modeling, animation and rendering package developed by the Autodesk Media and Entertainment Company [12]. According to [12] it is the world's largest selling 3D computer modeling and animation program. It is widely used by video game developers, TV commercial studios, architectural visualization or movie effects and pre-visualization. The 3D Studio Max contains a comprehensive set of tools for virtual world modeling, animation and rendering.

The most typical way for object modeling in 3DS Max is the polygonal modeling. The user starts from one of the predefined primitives and then modifies the object by applying some of the implemented modeling tools (e.g. extrude, bevel). 3DS Max offers the following

standard primitives: box, cylinder, torus, teapot, cone, sphere, tube, pyramid, plane or geosphere. Further, several extended primitives with more complex geometry are available as well.

Another advanced modeling method that is offered by 3DS Max is the NURBS modeling. This method produces smooth edges and eliminates the flat polygons and geometry errors of the polygonal method. The NURBS provide a mathematical apparatus for representation of freeform surfaces, e.g. car bodies or ship hulls. Those NURBS models can be exactly reproduced at any resolution when needed.

Besides the modeling capabilities, the 3DS Max offers variety of rendering options – either built-in or in the form of a plug-in. In addition, common tools such as scripting, texture editing, key-framing, skinning or inverse kinematics are also provided by the application.

3.2 Systems for Visualization of Virtual Worlds

Systems for visualization of virtual worlds specialize at enabling the user to explore the virtual worlds and to interact with them in some cases. Hence, the priority of such systems is the speed of the visualization. The realism of the displayed scene is sacrificed to this goal. The requirements on such systems are [5]:

- The visualization runs in real-time and the virtual world immediately responds to the actions of the user.
- The virtual world is 3-dimensional or it provides an illusion of being 3-dimensional.
- The user can explore the virtual world by moving in it and interacting with it.
- The virtual world does not have to be static. It can interact with the user. Objects can be manipulated or they are animated.

One of the most common languages used for description of virtual world is the *Virtual Reality Modeling Language* (VRML). The following section gives an overview of the features of VRML.

3.2.1 VRML

The first version of the Virtual Reality Modeling Language (VRML) – VRML 1.0 was defined by the Silicon Graphics Company in 1995. Their main motivation was to introduce a format that would be suitable for visualizing virtual scenes on the World Wide Web. Its second version – VRML 2.0 or VRML 97 was formulated in 1997 and in the same year it became an ISO standard under the number ISO/IEC 14772-1:1997 [10].

The main contribution of introducing this standard was an important change in the way that graphical data (virtual worlds) could be presented on common personal computers. Previously, the visualization of virtual data was a domain of expensive and highly specialized systems such as CAD. With VRML the virtual worlds could be explored by anyone on a common computer. Furthermore, its orientation on web enabled anyone who was connected to the network to view, explore and interact with the available virtual worlds. The fundamental features of VRML can be summarized in the following several points [10]:

- The virtual worlds are composed of combination of objects and multimedia elements such as images, video, and sound.

- Virtual worlds can use elements stored locally or anywhere on the web.
- The static worlds can be easily extended to dynamic worlds by adding animations, interaction and manipulations.
- The language can describe the movement and interactions of the user in the environment.
- VRML-based worlds can be inserted into web pages.
- VRML can be combined with other languages (Java, JavaScript).
- VRML language is stored in a text format.

#VRML V2.0 utf8	- <i>VRML file header</i>
WorldInfo {...}	- <i>Information about the virtual world</i>
Viewpoint {...}	- <i>Sets the viewpoint in the world</i>
Transform {...}	- <i>Description of the objects in the world</i>
Group {...}	
PositionInterpolator {...}	
ROUTE ... TO ...	- <i>Routing of the interactions in the scene</i>

Fig. 12 The structure of the VRML file.

Structure of the VRML File

The VRML file contains several logically separated parts, which describe different components of the virtual world. The file typically starts with a header that states what version of VRML and what textual encoding is being used. After that, the information about the virtual world itself and the definitions of viewports follow. The next part, which is typically the most extensive one, defines the geometry of the virtual world. Here, the static as well as the dynamic objects are defined. Finally, the routing interconnections between static and dynamic objects are stored in the file. Fig. 12 shows the structure of the VRML file.

The VRML language defines a graph of the scene. Nodes in this graph represent objects that should be handled. Grouping nodes represent internal vertices. VRML defines several types of standard nodes that can be used to define anything from geometry and object appearance to nodes for describing the light sources. The VRML also allows for creating new nodes, called prototypes.

Each node has its specific name (e.g. *Box*, *Color*, *Sound*, *SpotLight*, etc.). Besides its name, each node contains specific fields that define certain attributes of the object stored in the node. For instance, the radius of a sphere can be defined or the intensity of the light source can be set. Each specification field contains the type, name and the default value of the attribute. Fields defined as *fields* are private and cannot be changed. Fields defined as *exposedFields* are public and can be modified by other nodes.

In addition to the specific fields, every node has a set of associated events that it can receive and send out. Nodes can receive number of incoming events marked as *eventIn*.

Similarly, nodes can generate multiple events marked as *eventOut*. As an example consider the output event *touchTime* generated by a touch sensor. This event can then be routed to the *startTime* input event of the time sensor, which can start an animation.

In the following text, nodes that are important for this work are described in more detail. This information was mostly extracted from the VRML 2.0 standard [13].

```
Viewpoint {
    eventIn      SFBool      set_bind
    exposedField SFFloat     fieldOfView 0.785398
    exposedField SFBool      jump        TRUE
    exposedField SFRotation   orientation 0 0 1 0
    exposedField SFVec3f      position    0 0 10
    field        SFString     description ""
    eventOut     SFTIME       bindTime
    eventOut     SFBool       isBound
}
```

Fig. 13 Viewpoint node.

Viewpoint

The viewpoint node defines the location and the orientation of the point from which the user can see the virtual world. This node is of a great importance for this thesis, because it is equivalent to setting a camera for scene rendering. The *position* field determines the location of the viewpoint in the scene. The *orientation* field contains a rotation transformation for adjusting the camera's orientation. The *fieldOfView* field represents the maximum viewing angle of the scene. The *jump* and the *description* fields are used for controlling the transition between viewpoints and for storing the textual description of the viewpoint. The *set_bind*, *bindTime* and *isBound* fields are used for determining or controlling if the current viewpoint is selected. The syntax of the viewpoint node is presented in Fig. 13.

```
Shape {
    exposedField SFNode appearance NULL
    exposedField SFNode geometry    NULL
}
```

Fig. 14 Shape node.

Shape

The shape node is used for creating rendered objects in the virtual world. Its syntax is given in Fig. 14. The appearance node specifies the visual properties of the shape object such as the material or texture. The geometry node specifies the actual geometry shape of the object.

Appearance

The appearance node is used to define the visual properties of particular geometry node. Its syntax is denoted in Fig. 15. The *material* field defines another material node with detailed specification of the properties of the material, which is used for rendering. The *texture* field must contain either an *ImageTexture*, a *MovieTexture* or a *PixelTexture* node.

```

Appearance {
    exposedField SFNode material          NULL
    exposedField SFNode texture          NULL
    exposedField SFNode textureTransform NULL
}

```

Fig. 15 Appearance node.

Finally, the *textureTransform* field defines a set of transformations that are applied to the texturing coordinates.

Material

The material node specifies the surface material properties for the given geometry. Those settings are then used by the VRML rendering engine. The syntax of the material node is shown in Fig. 16. The meaning of the *ambientIntensity*, *diffuseColor*, *specularColor* and *shininess* fields follow the description of the Phong's reflection model described in section 2.3.5. The *emissiveColor* field enables the user to create glowing objects. The *transparency* field is responsible for determining the clearness of the object or how much light passes through it.

```

Material {
    exposedField SFFloat ambientIntensity    0.2
    exposedField SFColor diffuseColor        0.8 0.8 0.8
    exposedField SFColor emissiveColor        0 0 0
    exposedField SFFloat shininess           0.2
    exposedField SFColor specularColor        0 0 0
    exposedField SFFloat transparency         0
}

```

Fig. 16 Material node.

The *geometry* field of the Shape node can contain one of the VRML nodes for describing the geometry of an object. The most versatile one is the *IndexedFaceSet* node.

IndexedFaceSet

The *indexedFaceSet* constitutes the most general VRML construction for defining the geometry of an object. This node provides the means for creating an arbitrary shape that is composed of elementary polygons. The most important fields are the *coord* and the *coordIndex* fields. The *coord* field contains a *Coordinate* node, which stores a set of coordinate points. The *coordIndex* field defines which coordinate points should be connected together to form a polygon. Similarly the *color* and the *colorIndex* fields are used for specifying the colors and indexing them. If the *colorPerVertex* field is set to FALSE, then the indexed colors are applied to each face. In the opposite case the colors are applied to each vertex. Fields *normal* and *normalIndex* are then used for specifying the normals of each polygon. In case that the *normal* field is set to NULL, the VRML engine will generate automatic normals. The value of the Boolean field *normalPerVertex* determines if the indexed normals will be applied to vertices or to polygons. The *texCoord* and *texCoordIndex* fields are used for determining the texturing coordinates of polygons. Field *ccw* specifies the direction in which the vertices of a polygon are stored. Field *creaseAngle* determines how normals

```

IndexedFaceSet {
  eventIn      MFInt32  set_colorIndex
  eventIn      MFInt32  set_coordIndex
  eventIn      MFInt32  set_normalIndex
  eventIn      MFInt32  set_texCoordIndex
  exposedField SFNode   color          NULL
  exposedField SFNode   coord          NULL
  exposedField SFNode   normal         NULL
  exposedField SFNode   texCoord       NULL
  field        SFBool   ccw            TRUE
  field        MFInt32  colorIndex      []
  field        SFBool   colorPerVertex TRUE
  field        SFBool   convex          TRUE
  field        MFInt32  coordIndex      []
  field        SFFloat  creaseAngle     0
  field        MFInt32  normalIndex     []
  field        SFBool   normalPerVertex TRUE
  field        SFBool   solid           TRUE
  field        MFInt32  texCoordIndex   []
}

```

Fig. 17 IndexedFaceSet node.

should be smoothed over the shared vertices. The *solid* field says if the shape is supposed to enclose a volume and thus it can be used for back face culling.

The syntax of the IndexedFaceSet node can be seen in Fig. 17. For specification of the *Coordinate*, *Normal*, *TextureCoordinate* and *Color* nodes, refer to the VRML 2.0 standard specification [13].

Transform

The transformation node works as a grouping node that defines a set of transformations, which are applied to all its children. The *translation*, *rotation* and *scale* fields contain common transformation descriptions. The rotation transformation can be performed around an arbitrary center point, which is specified by the *center* field. The *scaleOrientation* transformation is a rotation transformation that is applied only to the scaling operation. The *children* field contains a set of nodes that will be affected by transformations defined in this node. Finally, the *bboxCenter* and *bboxSize* are optional fields defining a bounding box

```

Transform {
  eventIn      MFNode   addChildren
  eventIn      MFNode   removeChildren
  exposedField SFVec3f   center          0 0 0
  exposedField MFNode    children        []
  exposedField SFRotation rotation       0 0 1 0
  exposedField SFVec3f   scale           1 1 1
  exposedField SFRotation scaleOrientation 0 0 1 0
  exposedField SFVec3f   translation     0 0 0
  field        SFVec3f   bboxCenter      0 0 0
  field        SFVec3f   bboxSize        -1 -1 -1
}

```

Fig. 18 Transform node.

around the children of this node. They might be used for optimization of the display process. The syntax of the *Transform* node is specified in Fig. 18.

The VRML implements a predefined sequence of individual transformations. Therefore, the order in which the transformations are specified within the node does not matter. This ordering can be only changed by a nested sequence of transform nodes. Given transformation matrices *C* (*center*), *SR* (*scaleOrientation*), *T* (*translation*), *R* (*rotation*), *S* (*scale*), the transformation of point *P* will be performed as follows:

$$P' = T \times C \times R \times SR \times S \times -SR \times -TC \times P \quad (3.1)$$

PointLight

This node specifies an omni-directional point light source. The position of the light is set by the *location* field. The color of the emitted light and its intensity are determined by the *color* and *intensity* fields. The light can be turned on or off by setting the *on* field. The point light source reaches the geometry within the distance defined by the *radius* attribute. The *ambientIntensity* field determines the intensity of the emitted ambient light. The *attenuation* field specifies the attenuation of the light. The value of the attenuation factor can be computed as follows:

$$atten = \frac{1}{(attenuation[0] + attenuation[1] \cdot r + attenuation[2] \cdot r^2)} \quad (3.2)$$

The syntax of the *PointLight* node is defined in Fig. 19.

```
PointLight {
  exposedField SFFloat  ambientIntensity  0
  exposedField SFVec3f  attenuation      1 0 0
  exposedField SFColor   color           1 1 1
  exposedField SFFloat   intensity       1
  exposedField SFVec3f   location        0 0 0
  exposedField SFBool    on              TRUE
  exposedField SFFloat   radius          100
}
```

Fig. 19 *PointLight* node.

SpotLight

This node implements the spot light source, which emits light from a specific point in a specific direction and it is constrained by an angle. The meaning of fields *ambientIntensity*, *attenuation*, *color*, *intensity*, *location*, *on* and *radius* is identically with the *PointLight* node. The direction of the spot light is defined by the *direction* field. Fields *beamWidth* and *cutOffAngle* determine the actual shape of the beam of light. The syntax of the *SpotLight* node is specified in Fig. 20.

Node Instancing

In order to reduce the size of VRML text files, a single VRML node can be referenced multiple times. This is called instancing and it is enabled by using the VRML code words DEF and USE. The DEF codeword defines the name of a certain node and creates the node in

```

SpotLight {
  exposedField SFFloat  ambientIntensity  0
  exposedField SFVec3f  attenuation      1 0 0
  exposedField SFFloat  beamWidth        1.570796
  exposedField SFColor  color             1 1 1
  exposedField SFFloat  cutOffAngle       0.785398
  exposedField SFVec3f  direction         0 0 -1
  exposedField SFFloat  intensity         1
  exposedField SFVec3f  location          0 0 0
  exposedField SFBool   on                TRUE
  exposedField SFFloat  radius            100
}

```

Fig. 20 SpotLight node.

the scene. By calling the USE code word with the appropriate name, this node can be instantiated again, without actually duplicating its whole description in the VRML file.

The DEF code word is crucial for this project, because it enables labeling of various nodes by specific names that could be later used for recognizing the objects of interest.

VRML contains many additional types of nodes, which are not essential for this work. Their description can be found in the specification of the VRML standard [13]. For instance, there are nodes for manipulation with objects, sensors and interpolators, or nodes for creating a multiple levels of details of certain object.

3.2.2 3D Studio Max to VRML Conversion

The 3D Studio Max application contains a build-in exporter of the modeled scene into the VRML language. The scene can simply be saved into the VRML format. This proved to be of a great advantage for this project, as no additional implementation had to be done to achieve the data conversion. Furthermore, the export preserves the hierarchical relations in the modeled scene. In other words, the object hierarchies (grouping, transformation nodes) and their exact names will be transferred into the VRML format unchanged. Without this capability, it would be hardly possible to locate objects such as doors or windows in the exported VRML files and correctly modify their geometry.

3.3 Systems for Image Synthesis of Virtual Worlds

Systems for photo-realistic image synthesis implement sophisticated algorithms capable of simulating most of the complex physically based phenomena responsible for the illumination of a scene. Their absolute priority is the realism and quality of the produced image. Unlike the systems for visualization of virtual worlds, systems for image synthesis do not typically allow any interaction or exploration of the virtual world. The world can be only seen from a specified camera position and there are no dynamic parts in the world. The tradeoff for the high degree of realism of the produced images is substantial computational and memory requirements of the image synthesis process.

Most of the rendering systems for photo-realistic image synthesis are based on the ray-tracing algorithm. While the actual implementation may vary, those systems should implement at least the following objects and phenomena [16]:

- Cameras – The cameras define the viewpoint of the scene. The rays coming from the observer into the scene are generated through the camera.

- Ray-object intersection – The system has to be able to precisely determine the intersections of simulated rays with the geometry of the scene. Additional properties, such as surface normals, should be available.
- Light distribution – The system has to be able to describe and simulate various types of light sources and the way they distribute the light within the scene.
- Visibility – One of the most frequent tasks of the rendering systems is determining whether a given point is visible from a particular light source. This determines whether light from the light sources reaches the geometry or if it is occluded by other geometry.
- Surface scattering – At the ray-geometry intersection point the system should determine important properties of the surface that influence the way the rays of lights will be reflected or modulated.
- Recursive ray tracing – Tracing rays of lights after several bounces from shiny surfaces or refractions through materials such as glass, require implementation of the recursive ray-tracing. Also, a suitable stopping criterion should be implemented in order to maintain acceptable computational complexity.
- Ray propagation – When rendering optically active environments, such as smoke or atmosphere, the systems has to determine the modulation of the light as it travels through the space.

Table 1 Plug-in categories of the PBRT application.

Plug-In	Description
Shape	Provides access to geometric properties of the primitive such as its surface area, bounding box or ray intersection routines.
Primitive	Includes means for describing the geometry of the scene and its appearance.
Camera	Includes functions for camera definitions.
Sampler	Includes techniques for selecting sampling points of the scene.
Filter	Includes antialiasing techniques.
Film	Adjust the properties of the output image.
ToneMap	Includes methods for adjusting the dynamic range of the computed radiances.
Material	Includes class describing the material properties.
Texture	Implements textures.
VolumeRegion	Implements the optically active environments.
Light	Description of the light sources.
SurfaceIntegrator	Global illumination techniques.
VolumeIntegrator	Global illumination techniques in optically active environments.

In previous work, the *Physically Based Ray Tracer* (PBRT) was chosen as a suitable tool for this photorealistic image synthesis. This choice was mainly based on the capability of

PBRT to describe complex geometry, implement user-defined materials and use real light sources described by the goniophotometric diagrams. Furthermore, the input file format of the PBRT is very similar to the VRML file format, which simplified the data transformation process. The PBRT software is described in more detail in the next section.

3.3.1 PBRT

The *Physically Based Ray Tracer* (PBRT) is a photo-realistic rendering system, widely recognized by the computer graphics research community. It implements state of the art algorithms for photo-realistic image synthesis and their supporting data structures and functionalities. The three main goals for the design and implementation of PBRT were that it should be complete, illustrative and physically based [16].

One of many advantages of the PBRT system is that it was implemented as a plug-in architecture. The core of the PBRT drives the system's main flow but it does not contain any classes or functions related to specific elements of the rendering system. Plug-ins are loaded into the core as modules at run time. This makes the PBRT application easy to be extended and enhanced by new algorithms and functions. This is certainly one of the main reasons why PBRT became so popular in the research community. **Error! Reference source not found.** Table 1 lists all of the 13 plug-in categories of PBRT together with their brief description [16].

The PBRT system has three main phases of execution: input parsing, rendering and post-processing. During the first phase, the input file is parsed to load the description of the rendered scene. The input text file specifies the geometry of the scene, material properties, lights distribution and their types, definition of the camera and parameters for the rendering algorithms. In the second phase, the rendering loop is executed on the loaded data. The program spends most of its time in this rendering loop until the contribution of all lights samples are accounted for. In the final phase, the post-processing of the elapsed scene rendering is performed. This includes computing several statistics and preparing the system

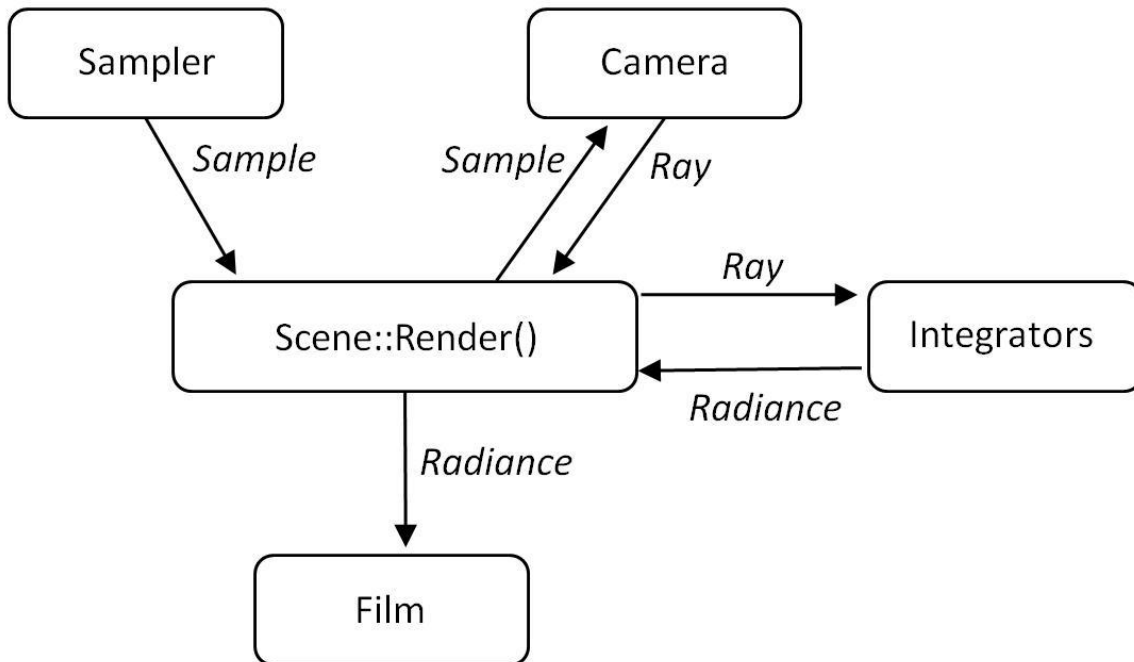


Fig. 21 PBRT main rendering loop [16].

```

SurfaceIntegrator "whitted" "integer maxdepth" [2]

LookAt 0 10 100 0 -1 0 0 1 0
Camera "perspective" "float fov" [30]

PixelFilter "mitchell" "float xwidth" [2] "float ywidth" [2]

Sampler "bestcandidate"

Film "image" "string filename" ["simple.exr"]
    "integer xresolution" [200] "integer yresolution" [200]

WorldBegin

AttributeBegin
    CoordSysTransform "camera"
    LightSource "distant"
        "point from" [0 0 0] "point to" [0 0 1]
        "color L" [3 3 3]
AttributeEnd

AttributeBegin
    Rotate 135 1 0 0
    Material "matte"
        "texture Kd" "checks"
    Shape "disk" "float radius" [20] "float height" [2]
AttributeEnd

WorldEnd

```

Fig. 22 Example of PBRT input file.

for a new rendering of another scene.

An illustration of the inter-module communication during the main rendering loop is shown in Fig. 21.

In the rest of this section, the syntax and semantics of the PBRT input files will be explained. A complete documentation to the PBRT application and its input file format can be found in [16].

Input File Format

The PBRT input files are written in a textual format. Fortunately, the format is relatively similar to the VRML input file format, which is of a great advantage for this project. An example of a scene written in the PBRT file format is given in Fig. 22. At the top of the file, the specification of the algorithm used for rendering and its parameters can be found. It is followed by specification of the camera, its type, position, orientation and field of view. Next the pixel filter used for anti-aliasing and the sampler are specified. The header is concluded by determining the film, which means setting up the name of the output file and the resolution of the output image.

The body of the PBRT file is introduced with a code word *WorldBegin*. In this section various objects can be defined. In the shown example, a light source is created first. It is followed by the scene geometry, which consists of a disk primitive. Notice the similarity with VRML when the transformations are preceding the definition of the appearance and the geometry of the object. Objects are introduced with a code word *AttributeBegin* and finalized

```

...
ObjectBegin "Repeating_Element"
    Material ...
    Shape ...
Object End

...

AttributeBegin
    Rotate ...
    Translate ...
    ObjectInstance "Repeating_Element"
AttributeEnd
...

```

Fig. 23 PBRT object instantiation.

with a code word *AttributeEnd*. By nesting multiple attributes within each other a scene graph can be constructed as is the case with the grouping nodes of VRML. The scene definition is closed with the code word *WorldEnd*.

Object Instanting

In order to reduce the size of the input file, PBRT offers a method for object instancing. In other words, the geometry of repeating object can be defined in one place in the file and the object can be instantiated multiple times. An example of object instantiation is shown in Fig. 23. The material and shape of the object is first defined in the object definition. Then by calling the code word *ObjectInstance* and selecting the appropriate name of the object, a specific object can be instantiated.

This is very similar to the VRML object instancing using the DEF and USE construction. However, there is one important difference. Object definition in PBRT does not insert the object into the scene yet. This is only done by calling the instance of the object. In contrary, the VRML object definition itself adds the object into the scene.

Include

Another convenient feature of the PBRT file format is that it supports modular structure of the data set. This means that the input data can be stored in multiple files, which can be linked together. This can be performed by calling a codeword *Include* and the file path of the input file being included.

Again, this is very beneficial for this project, because the modularity of the data stored in VRML can be preserved after transformation into the PBRT format. This means that different layers of the MPII model can be processed separately. The geometry of all of the used layers can then be included into a single main file.

Geometry

The PBRT enables geometry specification of both basic primitives as well as for complex geometric elements such as NURBS surfaces. The following shapes can be used: cone, cylinder, disk, hyperboloid, loop subdivision surface, NURBS surface, paraboloid, sphere or triangular mesh.

```

...
AttributeBegin
  Material ...
  Shape "trianglemesh"
    "point P" [0 0 0 1 0 0 1 0 1 0 0 1]
    "integer indices" [0 1 2 2 3 0]
AttributeEnd
...

```

Fig. 25 PBRT triangular mesh.

For the purpose of this work, the triangular meshes are the most important ones. They are the PBRT counterpart of the VRML indexed face set used for describing the geometry of the MPII data set. The triangular mesh specifies a set of points and their indices that create triangles. In addition, the specification of per vertex normals, tangents and texture coordinates is optional.

An example of the geometry specification using the triangular mesh is presented in Fig. 25. Here, four points are defined forming a square. These points are then connected using the provided indices to form two triangles.

```

...
AttributeBegin
  Translate ...
  LightSource "goniometric"
    "string mapname" ["IESsiemens.exr"]
    "color I" [33000 32511 28743]
AttributeEnd
...

```

Fig. 24 PBRT photo-goniometric light source.

Light Sources

PBRT supports all of the commonly used light sources. The list starts with the basic ones such as point lights and spot lights. To achieve realistic illumination of the scene, PBRT provides area lights. An unusual light source is the projection light, which enables projection of a texture on the scene, thus achieving the effect of projecting an image on a wall. Distant lights can be used for simulating illumination coming from a long distance, when the beams of light become parallel. In order to simulate the effects such as skylight or other environmental lighting the infinite area light is implemented. Here, environmental map can be used for scene illumination. Probably the most important type of light for this project is the goniophotometric light. This light allows for specifying a goniophotometric diagram that determines the distribution of the outgoing light. This type of light allows for using the previously measured goniophotometric diagrams for the lights sources in the MPII building.

An example of the PBRT format of the goniophotometric light is shown in Fig. 24. The name of the file with the diagram is specified together with the color of the light. Additional details of how goniophotometric lights are implemented using PBRT can be found in [1].

Materials

In PBRT each geometrical object maintains its material. The material defines how the light will interact with the surface upon intersection. PBRT offers a variety of material types, each of which can be further parameterized. This allows for a very accurate description of the appearance of the scene. The supported material types are as follows: blue paint, brushed metal, clay, felt, glass, matte, mirror, plastic, primer, shiny metal, skin, substrate, translucent and uber. The list of parameters for each type of material can be found in [16].

```
...
AttributeBegin
  Material "uber"
    "color Kd" [0.8 0.4 0.4]
    "color Ks" [0.9 0.9 0.9]
    "color Kr" [1.0 1.0 1.0]
    "float roughness" [0.05]
    "float opacity" [1.0]
  Shape ...
AttributeEnd
...
```

Fig. 26 PBRT uber material.

As an illustration, the syntax of the most general and powerful material uber is presented in Fig. 26. Parameters *Kd*, *Ks*, and *Kr* define the diffuse, glossy and specular reflection coefficients, respectively. Parameters *roughness* and *opacity* allow for additional parameterization of the material.

Rendering Algorithms

PBRT offers five basic surface integrators plug-ins. The two basic rendering algorithms are the whitted ray-tracer and the direct lighting surface integrator. Whitted ray-tracer follows the classic implementation of the ray-tracing algorithm. It computes direct lighting and it uses recursive ray-tracing for obtaining the specular reflection and refraction. The direct light surface integrator is based on the whitted ray-tracer. It allows for additional parameterization, such as choosing the strategy for sampling the light sources. In order to compute the indirect illumination, the path tracing, irradiance caching and photon mapping

```
SurfaceIntegrator "photonmap"
  "integer causticphotons" [0]
  "integer indirectphotons" [500000]
  "integer directphotons" [500000]
  "bool finalgather" ["false"]
  "integer finalgathersamples" [32]
  "integer nused" [50]
  "bool directwithphotons" [False]
  "integer maxdepth" [10]
  "float maxdist" [22.5]
```

Fig. 27 PBRT photon mapping surface integrator.

algorithms are implemented. These algorithms have been previously explained.

As an example consider the syntax of the photon mapping surface integrator illustrated in Fig. 27. The syntax allows for specification of the number of photons for each photon map. Also, the user can specify if the final gathering should be used and if so how many samples should be used. Other parameters specify if the direct lighting should be computed separately using the whitted ray-tracer or if the photon map should be used. An important parameter influencing the performance and time intensity of the algorithm is the maximum distance from photons where the algorithm can still use the cached values.

3.3.2 VRML to PBRT Conversion

The MPII data model was created using the 3DS Max modeling software. 3DS Max contains a build-in data exporter to the VRML language. The last remaining step is to convert the scene from VRML format into the PBRT input format. This has been already accomplished in previous work of Jiří Drahokoupil. A parser of the VRML format was implemented using lexical and syntactical analyzers. The parsed VRML data were loaded into the internal data structures. Next, the data can be exported into the PBRT format. Of a great help is the similarity between the VRML and the PBRT input file formats. Most importantly, both formats enable creating a hierarchical graph of the scene and they allow for object instancing. Transformations applied in one node of the hierarchy are applied to all of its children. Finally, the VRML definitions of geometry, materials and light sources all have their PBRT counterparts with almost identical parameters.

4 Previous Work

This thesis is a part of an ongoing project that was initiated at Max-Planck Institute of Informatics (MPII) in 2003. Its current state can be attributed to the effort of Josef Zajac and Jiří Drahekoupil and to the management and supervision of Dr. Vlastimil Havran. This chapter summarizes previous work that has been done up to this date.

4.1.1 Data Model

The effort to model the MPII building was initialized by Dr. Vlastimil Havran in 2003. Majority of the geometry was modeled by Josef Zajac in 2003 and 2004. The MPII building, built in 1995, was chosen for its obvious suitability as a data model for predictive image synthesis. Recalling the requirements for such a data model from section 2.1, the building had sufficient geometrical complexity, it contained complex lighting scenes with multiple light sources, over 60 various materials were present and most importantly the building itself was a research complex, thus available to the researchers for thorough documentation during the modeling process.

Initially, it was believed that available 3D data in DFX format would accelerate the modeling process. However, those turned out to be of a very little use to Jose Zajac as the data migration process turned out to be unsuccessful. The modeling effort continued based on obtained technical drawings and self-acquired measurements from the MPII building. The size and the complexity of the model soon required decomposition into separate layers. The model was finalized in 2004 and it contained about 11 million of triangles. However, several major problems persisted. First, the export module in the 3DS Max modeling software into the VRML language did not work properly. Second, the project was not completed as acquisition of the luminaries presented in the building was not performed and no BRDF for the materials in the MPII building was measured. Examples of the model created in 2004 can be seen in Fig. 28.

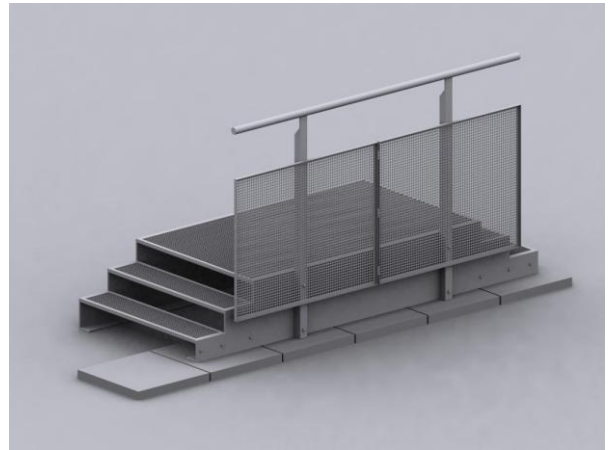
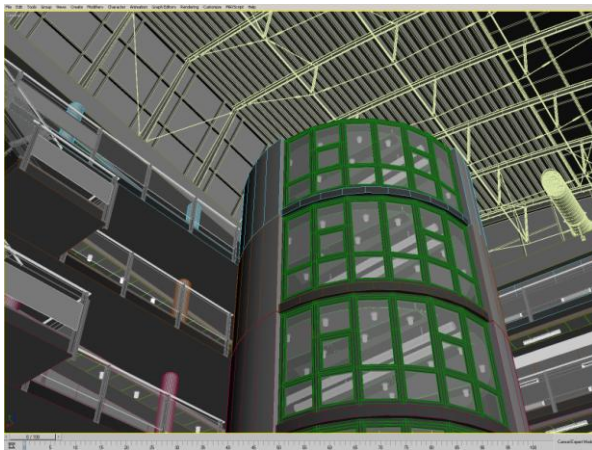


Fig. 28 Model of the MPII building constructed by Josef Zajac [1].

Due to the difficulties in resolving the above mentioned issues, the project was put on hold in 2004. In 2007, student Jiří Drahekoupil gained interest in the project in the scope of his Master Thesis titled “Predictive Image Synthesis from MPII Building Model”. The thesis consisted of three major tasks: i) completing the geometrical model, ii) measuring the actual luminaries and placing them into the model of the building, and iii) measuring or estimating

the BRDFs for the surfaces in the MPII building. Furthermore, Jiří Drahokoupil implemented a software application with graphical user interface that allowed working with the MPII data model and its export into different file formats. The software application is discussed in the next section.

The work on the geometrical model of the MPII building mainly consisted of documentation and modeling of additional equipment in the interior of the building, which was placed there after year 2004. Over 50 pieces of new equipment were modeled and added into the model. The fences on the main staircase were completely remodeled. Further, little trees on several floors were added. The furniture in the offices was made consistent with the real equipment of the building as of 2008. The final model, decomposed into multiple layers in 3DS Max contained over 40 million triangles.

In order to accurately measure the BRDF of a given material, a sample with dimensions 3cm x 3cm has to be acquired. Hence, the necessity to extract 63 samples of each material constituted an impossible task. The chosen solution was to approximate the properties of the materials based on their visual appearance known from photo-documentation. Several reflectance databases were used to find the most appropriate estimates. Mainly the Columbia-Utrecht Reflectance and Texture Database – CURET was used [8]. Appendix B list all the materials used in the MPII data model.

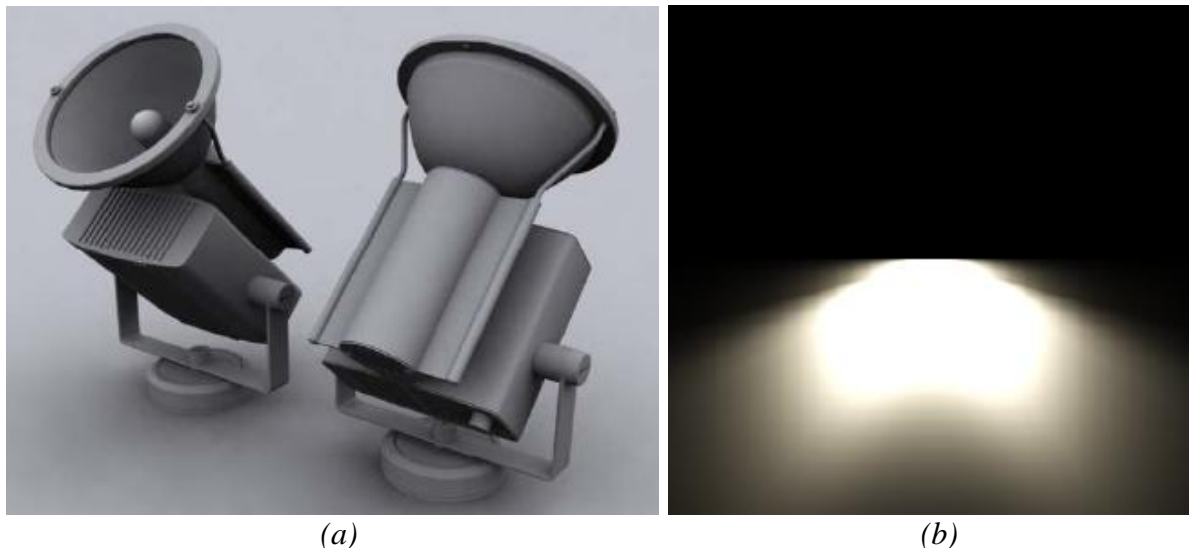


Fig. 29 Model of light (a) and its gonio-photometric diagram (b) [1].

The measurements of the luminaries were performed in cooperation with the Department of Electroenergetics at Czech Technical University in Prague. Exemplars of six lights presented in the building were transported to Prague from Saarbrücken. Because the lights were brand new they were first used for 100 hours. The measurements were done using a gonio-photometer device with rotating arm. The measured values were stored into the CIE IESNA file format. A conversion module for converting the IESNA format into a gonio-photometric diagram in the OpenEXR format was implemented. Example of a model of light and its gonio-photometric diagram can be seen in Fig. 29. An inconsistency between the labeling of individual light sources in the MPII model and the thesis of Jiří Drahokoupil was discovered. Appendix A now lists all the implemented light sources with their appropriate labels.

4.1.2 Implemented Application

Besides completing the data model, Jiří Drahoukoupil implemented a software application for data conversion. The major requirements for the application and functionalities that were implemented are as follows:

- The application can read an input VRML file exported from the 3DS Max modeling software.
- The positions, directions and types of light sources were modeled in 3DS Max and then maintained through the conversion into the VRML format. The application is capable of transforming the definitions of light sources into the PBRT format.
- The application can transform the CIE IESNA luminary description into the gonio-photometric diagram in the OpenEXR format.
- The user can specify properties of light sources in the model through the GUI of the application.
- Geometrical objects can be assigned appropriate material definitions defined in an input text file.

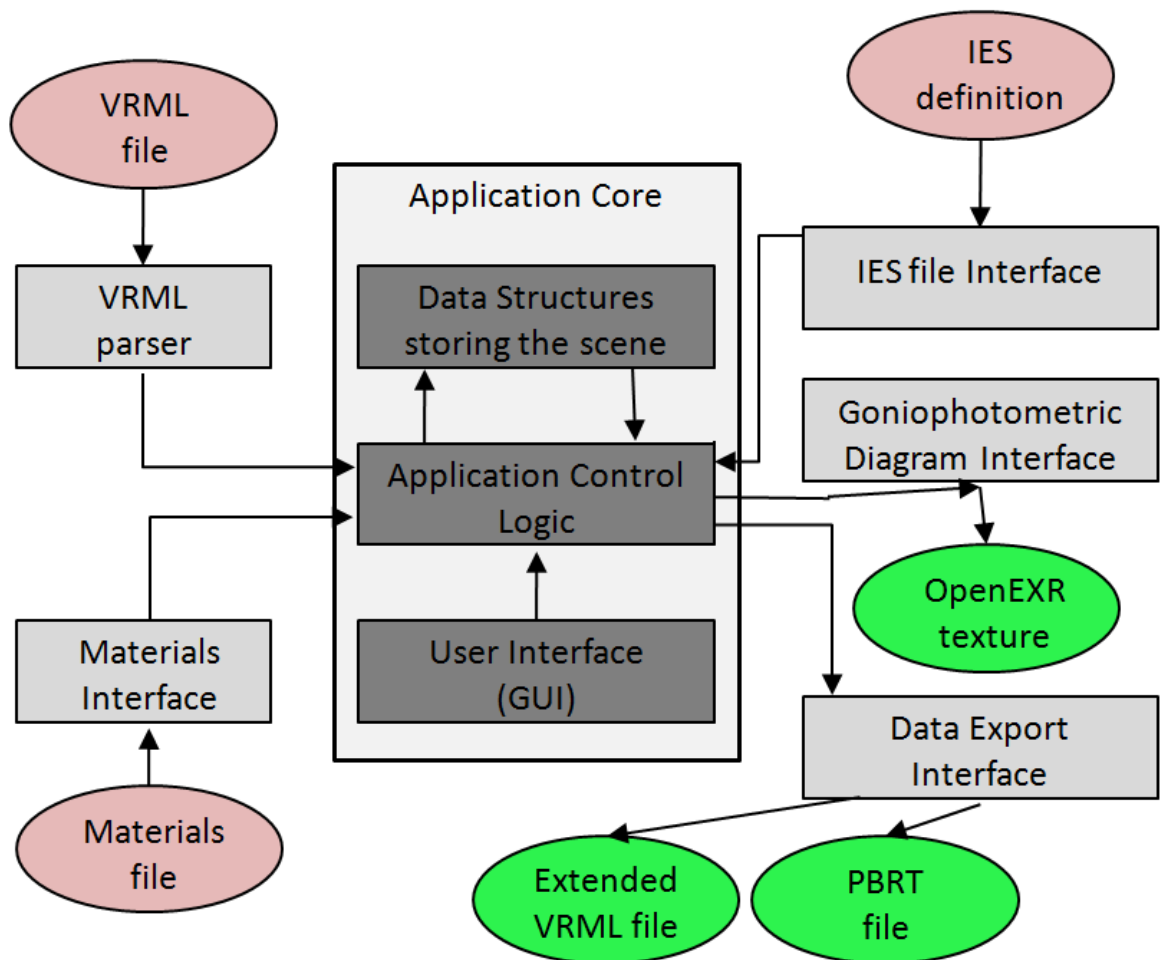


Fig. 30 Architecture of the implemented application for conversion of the MPII building data model [1].

- The position and properties of the camera, specified in the 3DS Max, will be maintained through the VRML to PBRT conversion. The user can specify which camera will be used for the PBRT image synthesis.
- The scene can be exported into an extended VRML format, which includes the specifications of luminaries and materials definitions.
- The applications should not be limited only to the specifics of the MPII model. Instead, it should constitute a general tool for enhancing and transforming similar datasets written in a predefined input format.
- The application supports working with multi-layered models. This is necessary for large-scale data models such as the MPII building model.

The architecture of the application is presented in Fig. 30. The core of the software maintains the data structures with the loaded model. It runs the control logic of the application, and it communicates with the user via GUI. Connected to the core are several modules implementing specific functionalities. VRML parser processes the input VRML file and stores it in the data structures of the core. Material interface takes the input text file with definitions of materials and stores them in particular data structures. The IES module can

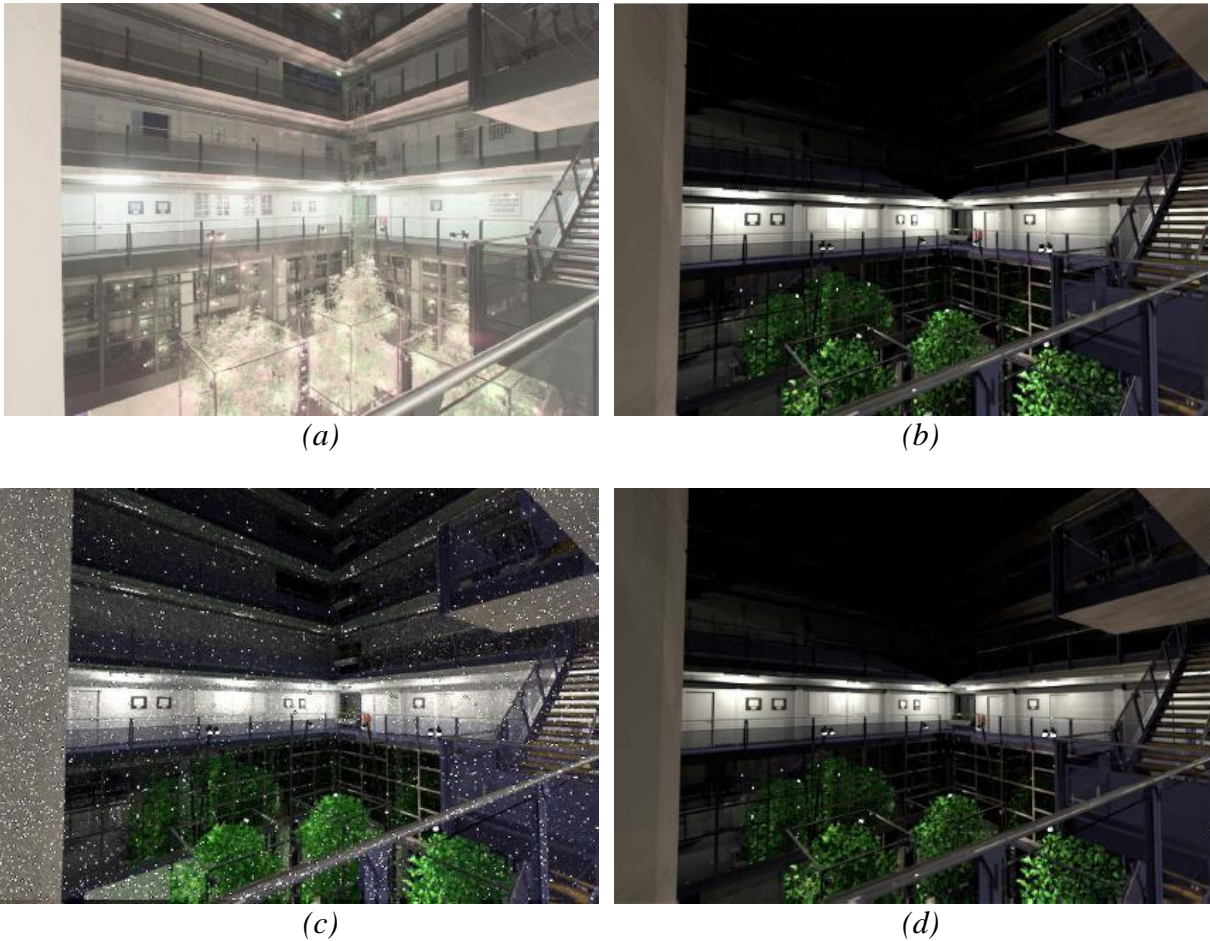


Fig. 31 Comparison of the results created by Jiri Drahokoupil. The real photograph is shown in (a). Images rendered using ray-tracer (b), path tracing (c) and instant global illumination (d) are shown next [1].

parse the input IESNA definitions. These can be turned into an OpenEXR texture via the gonio-photometric diagram interface. Finally, the interface for data export takes the data stored in the core data structures and writes them into the output text files in either the extended VRML format or the PBRT file format.

The graphical user interface was implemented using the wxWidgets library [9]. The development of this library has started in 1992 and it has been continuously extended since then. The application for conversion of the MPII model uses version 2.8.9. This wxWidgets library gives a simple access to all necessary elements of graphical user interfaces.

One of the biggest advantages of the implemented architecture is its modular structure. This enables relatively simple implementation of further extensions. New modules can be easily connected to the core of the application and work with the stored data. This turned out to be very important for the current work, as very little of the previous application had to be modified. Only new modules and functionalities were added.

4.1.3 Previous Results

In order to validate the correctness of the created data model and the functionality of the implemented application, Jiří Drahoukoupil presented several comparisons of the reference photographs taken in the MPII building and their corresponding virtual images synthesized from the data model using the PBRT software. Those results proved the correctness of the implemented work and showed the suitability of the whole MPII data model for predictive image synthesis.

Fig. 31 shows exemplary comparison of the real photograph obtained in the MPII building and the synthesized images from the PBRT software. Ray tracing, path tracing and instant global illumination algorithms were used for the image synthesis.

5 Analysis

This thesis builds on the conclusions, application design and data model architecture introduced in previous work [1]. Chapter 4 reviewed the history of the development and the state of the project up to this date. In order to accomplish the desired task of creating a dataset and supporting application tools suitable for predictive image synthesis, three main stages were implemented: i) data acquisition and modeling; ii) data-modification via software application; and iii) photo-realistic image synthesis based on the data model.

The important and crucial decisions regarding software tools used in this project had been already made prior to initiating work on this thesis. The 3DS Max (discussed in Section 3.1.1) was chosen as a suitable tool for creation of the data model and subsequent export into the VRML format. Also, the PBRT software was selected as the most appropriate candidate for carrying out the photo-realistic image synthesis.

The scope of this thesis is to analyze and design the means for implementing the geometry modifications of movable elements of the MPII data model (e.g. doors, windows, etc.) and for producing animations using the MPII data model.

5.1 Application Architecture

The data flow of the previously implemented application enables for loading an input VRML file, which was exported from the 3DS Max and for storing the geometry in the core data structures. Via several interfaces, this data can be merged with additional information such as materials and luminaries definitions. The application then allows for data export into the PBRT file format, which could be loaded into the PBRT application. The final results are rendered images synthesized by the PBRT application.

The task of this thesis is to implement the following two extensions:

- Geometry modification of movable objects in the MPII model. Those include doors, windows, shutters and auxiliary manipulation elements (e.g. door and window handles).
- Animation of movable objects, cameras and lights in the MPII model. The application would produce input data needed by the PBRT software for rendering such animations.

While these extensions do not require implementation of a new separate interface modules (such as the interface for materials or IES luminaries definitions), they require a substantial modifications and enhancements to the functionalities, data flow connections and user interface of the existing application.

Fig. 32 shows the proposed application architecture design. Rectangles, arrows and text fields in black color correspond to the previous version of the application. Red color marks the new extensions necessary for implementing the given tasks. Individual extensions are numbered and explained below.

1. Objects Descriptions – Movable objects (e.g. doors, windows) are exported into the VRML format as any other geometry in the scene. Therefore, a specific notation has to be introduced to enable a description of the objects of interest. This notation has to be correctly preserved during the conversion into the VRML format.

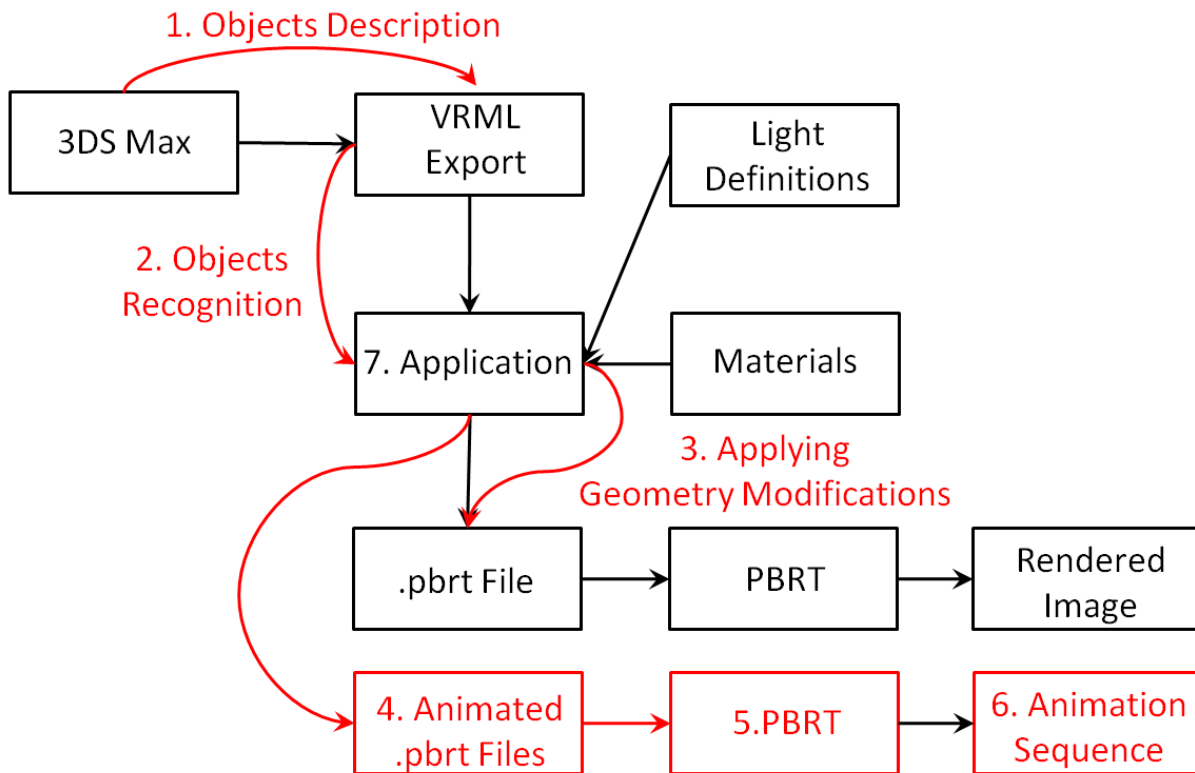


Fig. 32 Application architecture design.

2. Objects Recognition – The implemented application has to recognize the objects of interest based on the specific notation. The objects are presented to the user.
3. Applying Geometry Modifications – The previously implemented PBRT export module has to be updated to account for user-defined modifications of the movable parts of the geometry. The modifications applied to the VRML model have to be identically transformed into the PBRT model.
4. Animated PBRT Files – Upon user request, the application has to prepare the PBRT data model according the specifications of the animation. This should be done in an efficient manner in order to prevent working with unnecessarily large text files.
5. PBRT – The PBRT data model of an animation has to be rendered by the PBRT application. Since this might require a tedious and prolonged work from the user, a script should be automatically generated to simplify this task.
6. Animation Sequence – The output from the PBRT software is a set of rendered animation frames in high-dynamic range format. A suitable software tools enabling batch tone-mapping and merging multiple frames into a video sequence should be identified.
7. Application - In order to accommodate the above mentioned extensions, the core of the application itself has to be substantially modified. This mainly includes implementing new data structures, functionalities and GUI elements for working with the objects of interest and specifying the animation properties.

5.2 Geometry Modifications

The MPII model contains specifications of many cameras and light sources. These were correctly exported into their VRML counterparts by the build-in 3DS Max VRML export utility. Cameras were transformed into *Viewpoint* nodes and light sources were stored as *SpotLight* nodes. These objects were easily identified by the implemented VRML parser, because of their specific node types. All *Viewpoint* nodes in the VRML file corresponded to certain cameras in the 3DS Max model and all *SpotLight* nodes corresponded to certain light sources in the MPII model.

However, all of the movable objects of interest (e.g. doors, windows) are exported into VRML as any other part of the scene geometry. Their geometry is stored in *IndexedFaceSet* nodes, thus making them indistinguishable from any other geometry in the scene. The only possibility for identifying these objects is via introducing a naming convention, which would be transformed into VRML through the DEF construction. Furthermore, objects such as doors are complex groupings of simpler elements, for instance door handles, door joints, etc. These all should be identified in order to modify the geometry of the object correctly. Therefore, a specific object hierarchies and naming conventions for the objects of interest have to be defined.

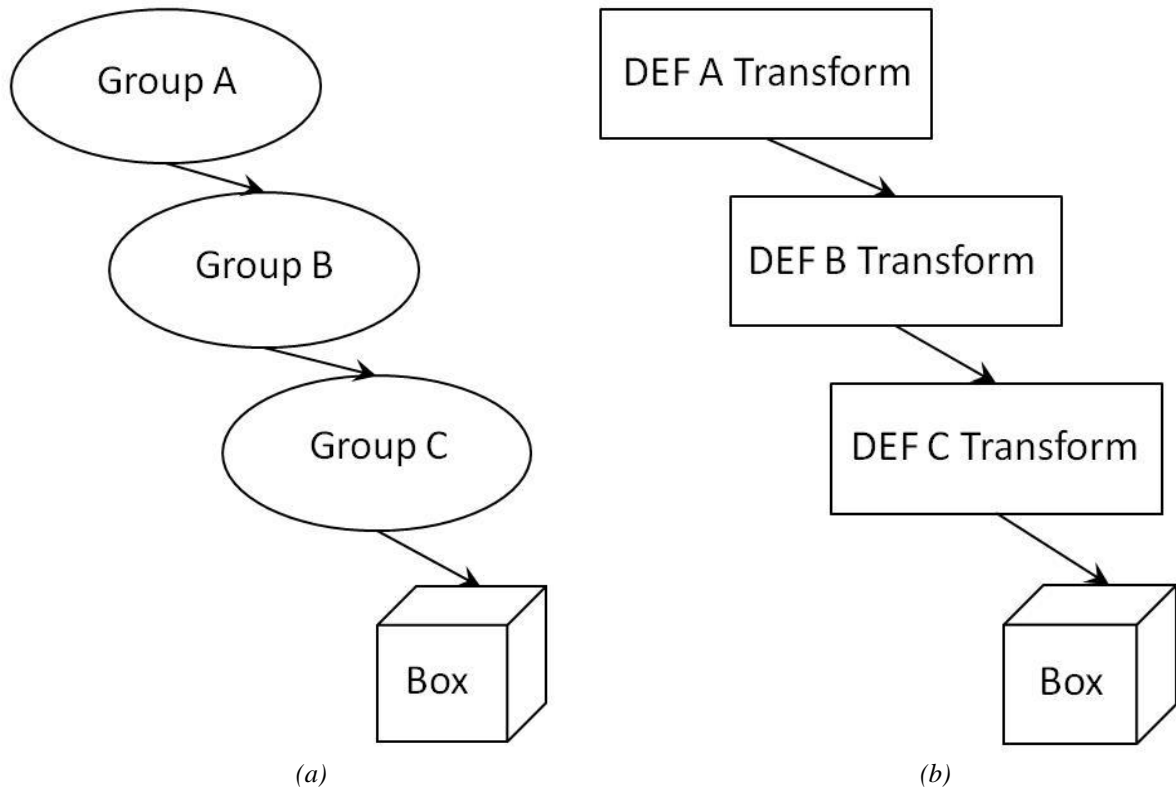


Fig. 33 Object hierarchy in 3DS Max (a) and after export into VRML (b).

5.2.1 Data Model

Initial analyses and examination of the MPII data model revealed that the data model was not constructed with the notion that geometry modifications of objects such as doors or windows might be implemented later. Various parts of one object – handles, keyholes, joints, doors, were located in different parts of the scene hierarchy. More importantly, the naming

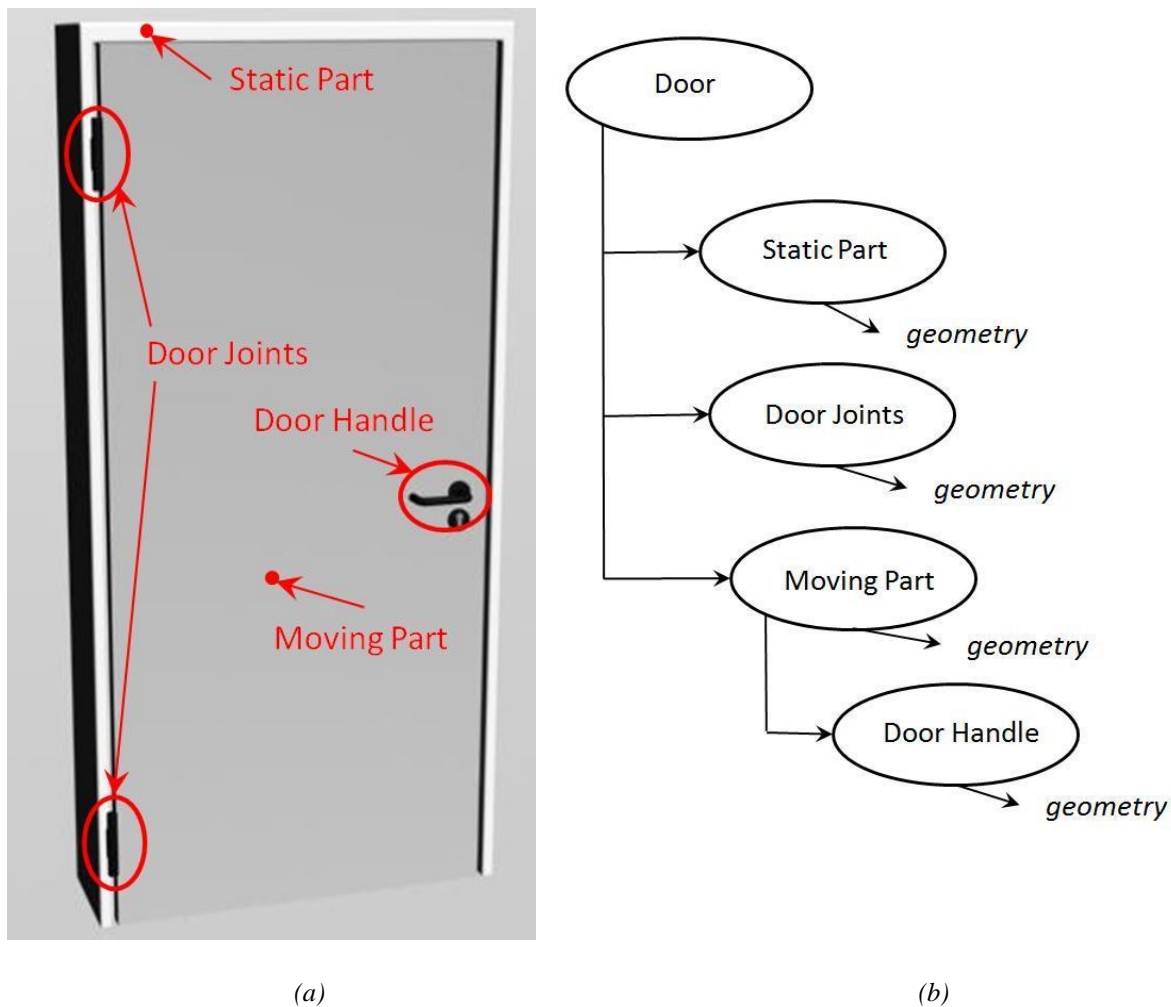


Fig. 34 Model of the door object (a) and its corresponding 3DS Max model hierarchy (b).

conventions were inconsistent. Therefore, regrouping of the object hierarchies and their appropriate naming was necessary.

It is important to realize how the 3DS Max hierarchies and objects names translate into the VRML format. In 3DS Max multiple objects can be grouped into a single *Group* node. Each group or individual objects have their names. *Groups* from 3DS Max will be translated into VRML *Transform* nodes and their names as the DEF construction. Consider the 3DS Max object hierarchy shown in Fig. 33(a). It consists of three nested *Groups* containing a box geometry at the end. Each *Group* has its own name. The transformation of this hierarchy into VRML is demonstrated in Fig. 33(b). It can be observed that each 3DS Max *Group* level corresponds to a *Transform* node in VRML with the appropriate name (DEF construction). Hence, by parsing the VRML scene graph, objects can be identified by decoding that name of the VRML node. It is also important to recall that transformation defined in a certain VRML transform node will be applied to all its children.

The constructed object hierarchies must enable easy access to all important parts of the object. Below, the 3DS Max hierarchies for each object of interest are designed and explained.

Door

Two kinds of geometry modification are to be performed with the door objects. First, the door can be opened. Second, the door handles can be manipulated. The action of door opening can be performed as a rotation of the moving parts of the object around an axis

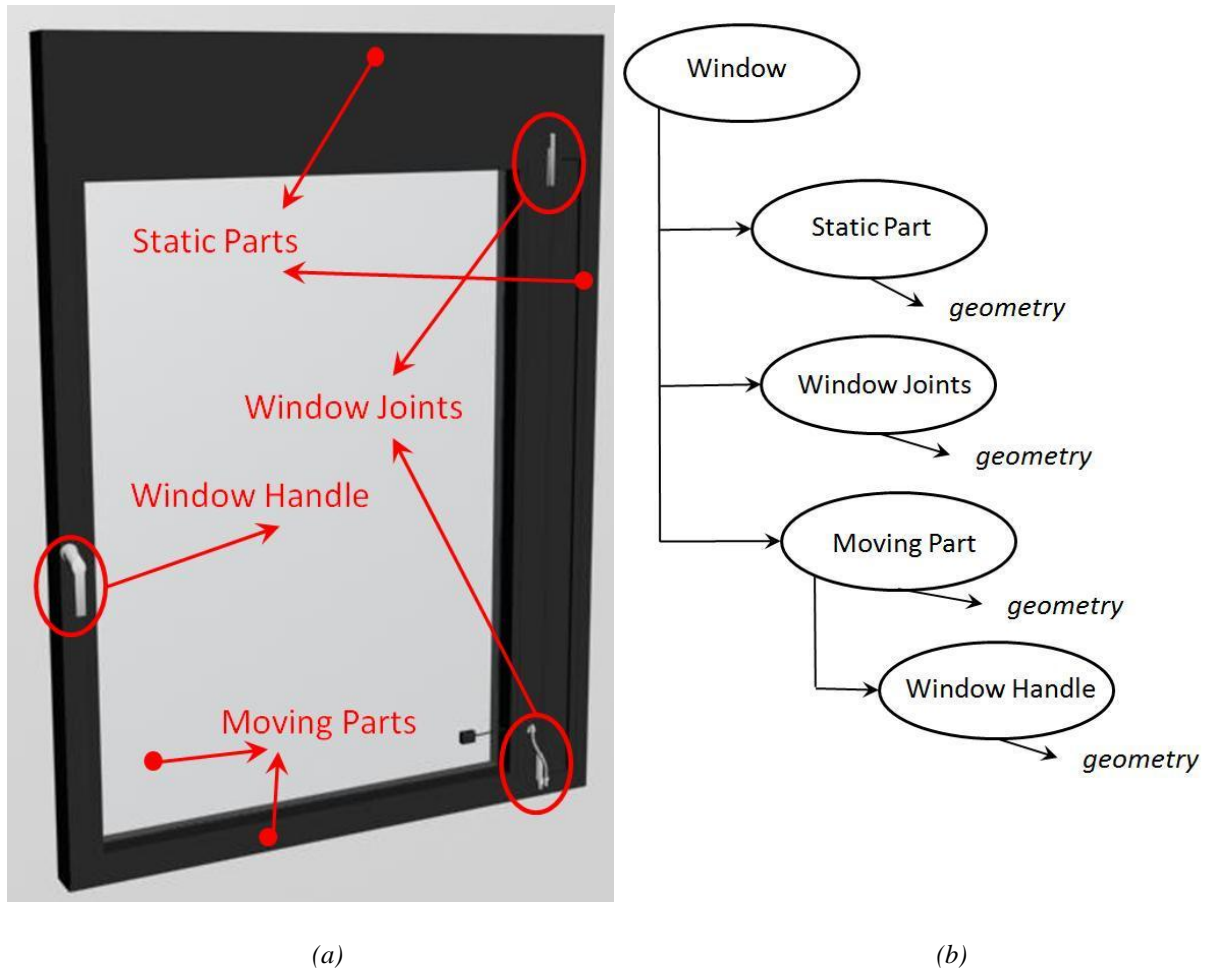


Fig. 35 Model of the window object (a) and its corresponding 3DS Max model hierarchy (b).

defined by the door joints. Door handles manipulation is also a rotation transformation applied to the door handles themselves. The door joints and static parts of the door object (e.g. door frame) should not be affected by the transformations.

Fig. 34(a) shows the 3DS Max model of the door object. Important parts of the object have been highlighted. In Fig. 34(b) the corresponding 3DS Max model hierarchy is displayed. It can be seen that access is enabled to all important parts of the model. Door joints can be identified and used for rotating the moving parts of the doors. This rotation will also affect the door handles. In addition the door handles can be rotated separately.

Window

The model and geometry of window are very similar to the door object, including the modifications necessary for allowing object animations. The moving parts of the window should be rotated around the axis defined by the window joints (both horizontal and vertical). In addition, the window handle should be opened to reflect its correct position when the window is opened in the real world.

Fig. 35(a) shows the 3DS Max model of the window object with significant components highlighted. The proposed hierarchy can be seen in Fig. 35(b). Again, it can be observed that the design of the hierarchy enables identification of the window joints and the

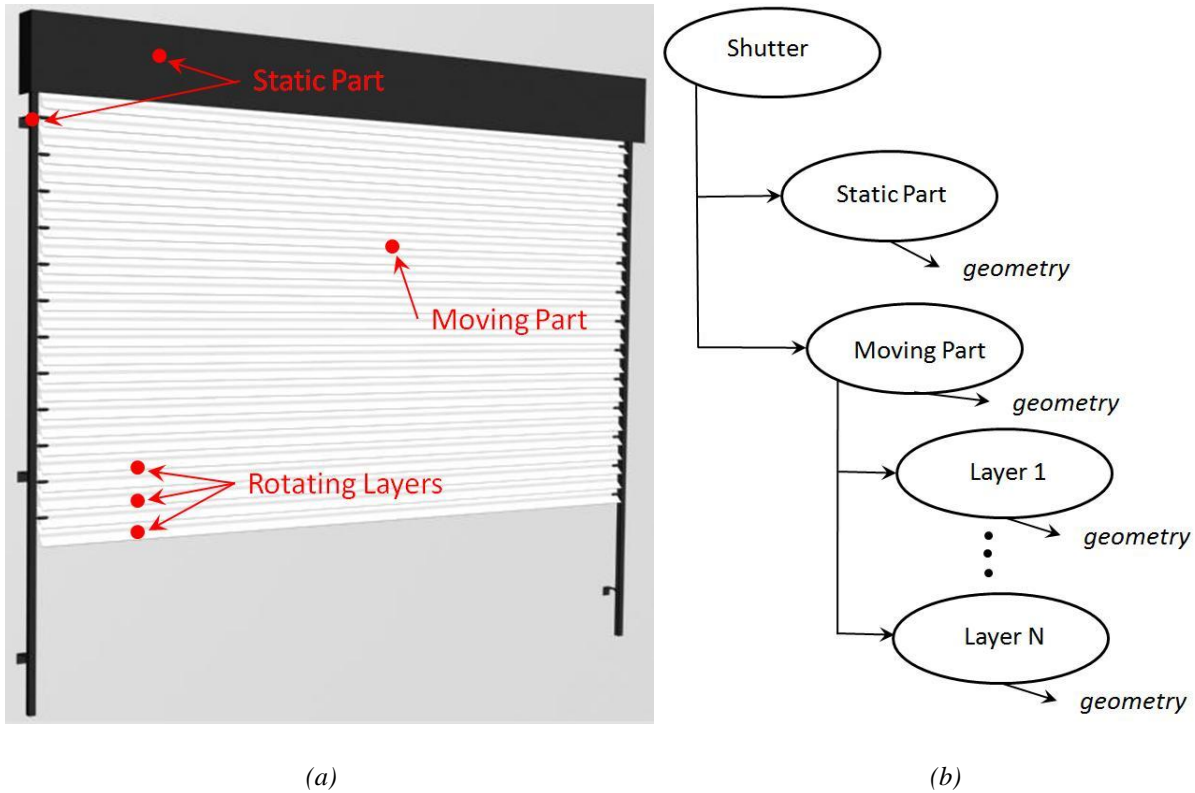


Fig. 36 Model of the shutter object (a) and its corresponding 3DS Max model hierarchy (b).

consequent correct rotation of the moving parts of the window. Also the window handles can be located and rotated accordingly.

Shutters

The shutter object is substantially different from the door and the window objects. The geometry modifications required are moving the shutters up or down in the frame and opening or closing them via rotating individual shutter layers. Moving the shutters up and down required only grouping of the moving parts together. However, rotating each individual layer required separating every layer, so that it can be rotated around its own axis.

The 3DS Max model with highlighted important parts can be seen in Fig. 36(a). The corresponding object hierarchy is presented in Fig. 36(b).

5.2.2 Objects Description

Including the underground and the ground floors, the MPII building has 8 floors. There are several hundred doors, windows and shutters in this complex building structure. These have to be somehow labeled, so that the user can operate with only the desired objects. The GUI of the application should display this description of each identified object. Furthermore, several variations of particular object exist in the building. For instance, there are doors as shown in Fig. 34(a), rotational doors in the main entrance or double glass doors on each floor. Different approaches have to be applied in order for them to be opened correctly. Hence, the description of the object should also include specification of the construction type of particular object.

Here, the preservation of object names through the data conversion from 3DS Max model to VRML is very important. Object hierarchies as discussed in the previous section, can be labeled in 3DS Max and this label would be translated into the DEF construction in the

VRML format. This label can encode the needed important information, such as location of the object (floor and section), its construction type or the identification number.

The previously implemented interface for describing light sources, which was implemented in previous work, was extended to accomplish this task. Originally, the lights were labeled in a specific format that enabled decoding the floor, section, type and the identification number of the light source. The new labeling of objects is designed as follows:

Code_Fx_Sy_Tz_Nww

The explanation of the individual parameters is given in Table 2 below.

Table 2 Explanation of individual parameters of the object labeling format.

Parameter	Explanation
Code	Code word specifying the object of interest. It can be one of the following: "Light", "Door", "Window", "Shutter"
Fx	Parameter F specifies the floor, where the object is located. Variable x stands for a number 0 - 9.
Sy	Parameter S specifies the section, where the object is located. Variable y stands for a number 0 - 9.
Tz	Parameter T specifies the construction type of the object. Variable z stands for a number 0 - 9.
Nww	Parameter N specifies the identification number of an object. Variable ww stands for a number 0 - 99.

As an example consider an object labeled as Door_F1_S2_T3_N42. By decoding the label it can be determined that this is a door object of construction type 3, which is located in the 2nd section on the 1st floor and its identification number is 42.

5.2.3 Object Geometry Modification

The geometry modifications are applied to the data model that was exported to VRML from the modeling software (3DS Max). Hence, the only mean for performing such geometry modifications is through applying a chain of transformations to isolated parts of the scene geometry (e.g. doors, windows). It is of a great advantage that both formats of describing a scene in VRML and PBRT define a graph of the scene, where transformations in one node are applied to all its children. Further, these transformations can be nested. As discussed earlier, the only difference between transformations in VRML and PBRT is that VRML has a hard-coded order of applying various transformations. Hence, this order also has to be enforced for the PBRT. Geometry modifications of each object of interest are discussed below.

Door

Recall the designed hierarchy of the door object from Fig. 34(b). Opening the door requires inserting a rotation transformation into the *Moving Part* node. The center of the

rotation will be determined from the position of the door joints. Manipulating with door handles requires inserting a rotation operation into the *Door Handle* node. The center of this rotation might be determined automatically by looking at the position of the handle base or by specifying a certain offset.

Window

Recall the hierarchy of the window object presented in Fig. 35(b). Two kinds of window opening should be implemented to reflect the real world situation. First, opening the window horizontally requires inserting a rotation transformation around the vertical axis of the window joints into the *Moving Part* node of the hierarchy. The center of this rotation is again determined by the location of the window joints. Second, opening the window vertically requires inserting a rotation transformation around the horizontal axis defined by the bottom of the window. The center of this rotation can be determined from the size of the window and its position. Finally, manipulating with the window handles requires inserting a rotation transformation into the *Window Handle* node. Again, the center of this rotation can be determined automatically by looking at the position of the handle base or by specifying a certain offset.

Window Shutter

Recall the hierarchy of the window shutter object shown in Fig. 36(b). The geometry modification of the window shutter object is very different from the door and the window objects. It should be possible to roll the shutters up and down and open them by rotating each layer. Rolling the shutters up and down can be implemented by inserting a scaling transformation into the *Moving Part* node of the shutter hierarchy. In case the center of the scaling transformation is in the middle of the shutter, the scaled object has to be shifted by applying additional translation to the top of the shutter frame. The length of this translation can be determined from the size of the shutter and from the applied scaling rate. Opening or closing the shutter requires inserting a rotation transformation around the horizontal axis to each *Layer N* node of the window shutter hierarchy. No center of this rotation needs to be specified, because the shutter layers are rotated around the center of the object, which is the default center of rotation.

5.3 Animations

An animation sequence is a composition of individual time-ordered frames. In context of this work, this means that the implemented software application should enable specification of the animation as well as generation of the input data for the rendering engine. The animation process requires several things to be implemented:

- User specification of global animation properties such as length of the animation or its frame rate.
- Access to individual parameters of the model that can be animated. This includes user interface, which will allow the user to specify the parameter values at certain time stamps.
- A global time generator, which will synchronize the animated objects and which will return the current time of the animation.

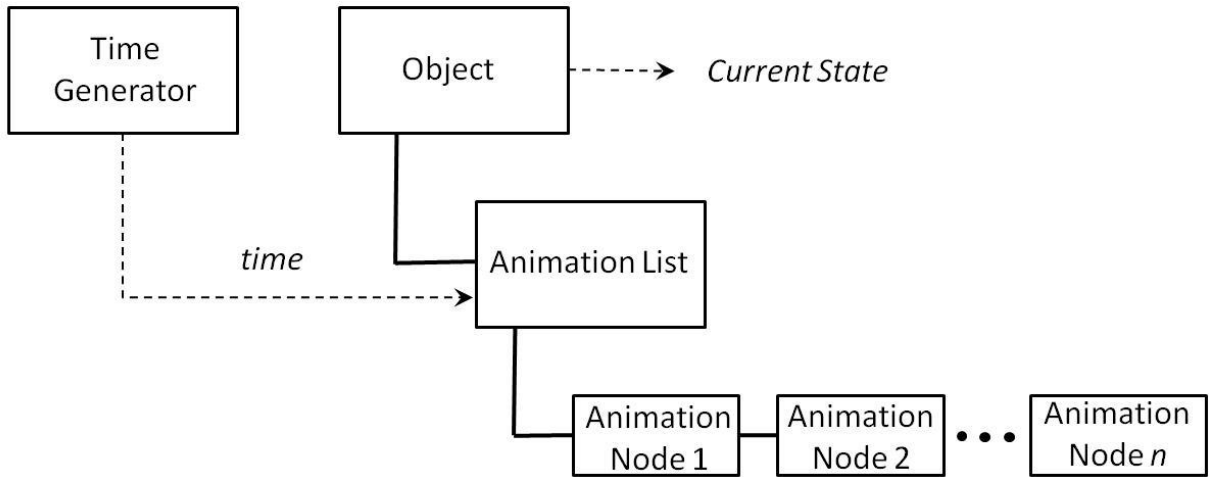


Fig. 37 Design of the animation workflow.

- Interpolators, which will calculate the current values of individual animated parameters based on the specification of the animation and the current time received from the time generator.
- Supporting functionalities for convenient generation of the input data files for PBRT.

The design of the animation architecture is presented in Fig. 37. Each object has a pointer to its animation list. The animation list holds a sequence of animation nodes ordered according to time. Each animation node stores its time and specific values of parameters of the object being animated. The global time generator object generates time events and sends them to all objects in the scene. During the generation of the input data files, the objects are asked for the current values of their parameters. These are returned and applied to modify the geometry or the state of the scene for the current frame.

5.3.1 Animation Nodes

While some parameters of the animation nodes are common to all of the objects (e.g. time of the animation), other parameters are only typical to specific objects. This suggested that a parent animation node should be implemented, which would encapsulate parameters shared by all types of objects. This animation node would serve as a parent for object specific animation nodes. It was required that the software application supports animation of lights sources, cameras and the movable objects in the building such as doors, window and window shutters. Parameters to be animated are listed below for each object.

Cameras

Cameras are exported from the 3DS Max as VRML *Viewpoint* nodes. The *Viewpoint* is defined by its position and an orientation vector. The orientation is determined by an axis and a rotation angle around this axis. Hence, the camera animation node should store the animated position and orientation of the camera.

Light Sources

Light sources are exported as spot lights from the 3DS Max and they are translated into a VRML *SpotLight* node. The implemented software application allows for turning specific lights on or off. The light animation node should also enable turning the light on or off at specific times during the animation. In addition the light intensity should be animated so that lights can be smoothly dimmed.

Doors

As described earlier, the door animation node has to store the animated open rate of the door and the open rate of the door handle.

Windows

The window animation node should be very similar to the door animation node. It has to store the current open rate of the window and the open rate of the window handle. In addition, since there are two possible ways of opening the window, horizontal and vertical, the animation node has to specify, which one is currently being performed.

Window Shutters

The window shutters animation node should store the roll up rate of the shutter object. Further, also the open rate of individual shutter layers has to be specified.

5.3.2 Interpolation

The animation nodes define the state of the scene (or individual objects) at certain time samples. In order to obtain the state of the scene in between the animation nodes some interpolation technique has to be applied. Animating simple geometrical objects such as the handles of doors or windows does not require sophisticated interpolation methods. The linear interpolation typically suffices in such cases. Animating more complex objects such as doors or windows requires more complex interpolation techniques that will resemble the way that people manipulate with them. Here, the linear interpolation can be substituted with the cosine or the cubic spline interpolation. Finally, the animation of the camera requires the most sophisticated approach. The desire is to compute a smooth trajectory of the camera based on predefined discrete points in the animation sequence. Further, there should be a mechanism for modifying the behavior and nature of the computed trajectory. A suitable candidate for this task is the Kochanek-Bartels spline (or TCB splines) [15].

In the following text, each of the four above mentioned interpolation techniques is discussed in more detail. Unless stated otherwise, all of the described functions take a and b as the input values, which are the values to be interpolated between, and parameter x which is a value between 0 and 1. Parameter x controls the interpolation process and determines the position between values a and b .

Linear Interpolation

Linear interpolation constitutes the simplest mean for interpolation between two values. A linear straight segment is inserted between each pair of neighboring points. Below is the pseudo-code of the linear interpolation function:

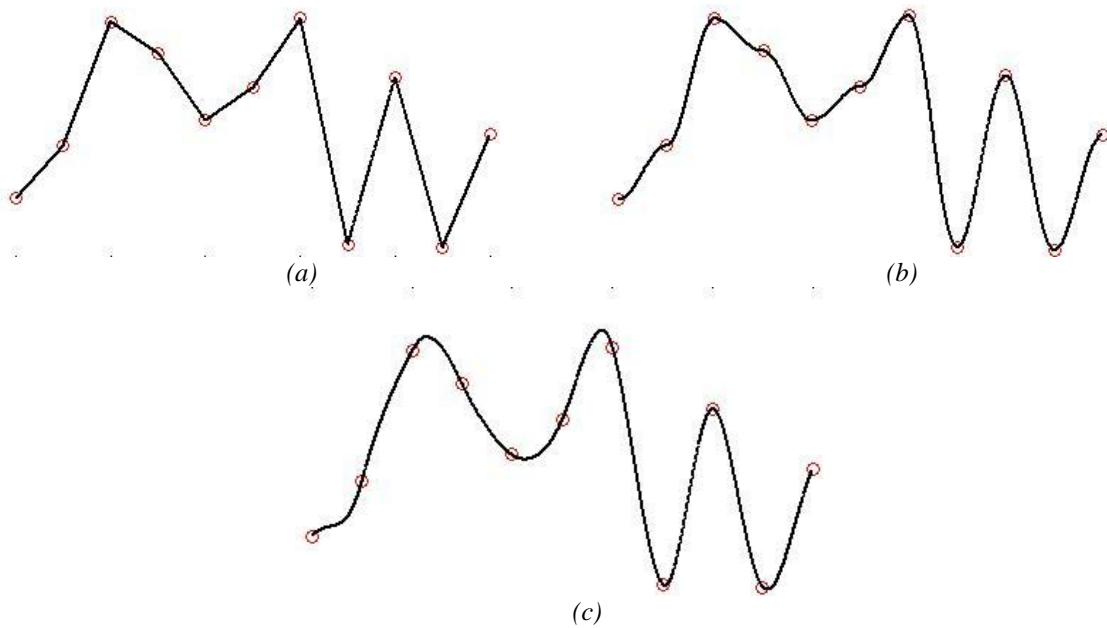


Fig. 38 Comparison of the linear (a), cosine (b) and cubic (c) interpolation methods on a set of points.

```
function Linear_Interpolate(a, b, x)
    return a*(1-x) + b*x
end
```

Cosine Interpolation

The cosine interpolation produces a substantially smoother result than the linear interpolation. This is achieved by substituting a modulated cosine function for the linear segments from the previous method. The pseudo-code of this method is shown below:

```
function Cosine_Interpolation(a, b, x)
    ft = x * 3.1415927
    f = (1 - cos(ft)) * 0.5

    return a*(1 - f) + b*f
end
```

Cubic Interpolation

The cubic interpolation method produces even smoother results than the cosine method. However, the price is a greater computational burden. Instead of looking only at points a and b , this method also works with points immediately to the left from point a and immediately to the right from point b . Those points are denoted as $v0$, $v1$, $v2$, and $v3$ in the left to right order. Below is the pseudo-code of the cubic interpolation method:

```
function Cubic_Interpolation(v0, v1, v2, v3, x)
    P = (v3 - v2) - (v0 - v1)
    Q = (v0 - v1) - P
    R = v2 - v0
    S = v1

    return P*x3 + Q*x2 + R*x + S
end
```

Fig. 38 compares the performance of the linear (a), cosine (b) and cubic (c) interpolation methods on a set points.

Kochanek-Bartels Cubic Splines

The Kochanek-Bartels Cubic Spline (KB Spline) is a cubic Hermite spline with build-in tension, continuity and bias parameters. These parameters modify the behavior of the tangents. A cubic spline formulated using the Hermite interpolation basis can be written as follows [15]:

$$X_i(t) = H_0\left(\frac{t-t_i}{\Delta_i}\right)P_i + H_1\left(\frac{t-t_i}{\Delta_i}\right)P_{i+1} + H_2\left(\frac{t-t_i}{\Delta_i}\right)\Delta_i T_i^O + H_3\left(\frac{t-t_i}{\Delta_i}\right)\Delta_i T_{i+1}^I \quad (5.1)$$

Here, P_i , T_i^I , T_i^O and t_i constitute the value, input and output tangents and time of the interpolation at point I and $\Delta_i = t_{i+1} - t_i$. The Hermite interpolation basis can be computed as follows:

$$H_0(s) = 2s^3 - 3s^2 + 1 \quad (5.2)$$

$$H_1(s) = -2s^3 + 3s^2 \quad (5.3)$$

$$H_2(s) = s^3 - 2s^2 + s \quad (5.4)$$

$$H_3(s) = s^3 - s^2 \quad (5.5)$$

Kochanek and Bartels computed the input and output tangents using the values of P_{i-1} , P_i and P_{i+1} and three parameters tension (t_i), bias (b_i) and continuity (c_i) for point i as follows:

$$T_i^I = \frac{(1-t_i)(1+c_i)(1-b_i)}{2}(P_{i+1}-P_i) + \frac{(1-t_i)(1-c_i)(1+b_i)}{2}(P_i-P_{i-1}) \quad (5.6)$$

$$T_i^O = \frac{(1-t_i)(1-c_i)(1-b_i)}{2}(P_{i+1}-P_i) + \frac{(1-t_i)(1+c_i)(1+b_i)}{2}(P_i-P_{i-1}) \quad (5.7)$$

The pseudo-code for the KB Spline interpolation method is given bellow:

```
function KBSpline_Interpolation(v0, v1, v2, v3, x, t, c, b)
    h0 = 2*x^3 - 3*x^2 + 1
    h1 = -2*x^3 + 3*x^2
    h2 = x^3 - 2*x^2 + x
    h3 = x^3 - x^2

    TDix = (1 - t)*(1 + c)*(1 + b)*(x2 - x1) / 2.0 +
            (1 - t)*(1 - c)*(1 - b)*(x3 - x2) / 2.0

    TSix = (1 - t)*(1 - c)*(1 + b)*(x3 - x2) / 2.0 +
            (1 - t)*(1 + c)*(1 - b)*(x4 - x3) / 2.0

    Return h0*x2 + h1*x3 + h2*TDix + h3*TSix
end
```

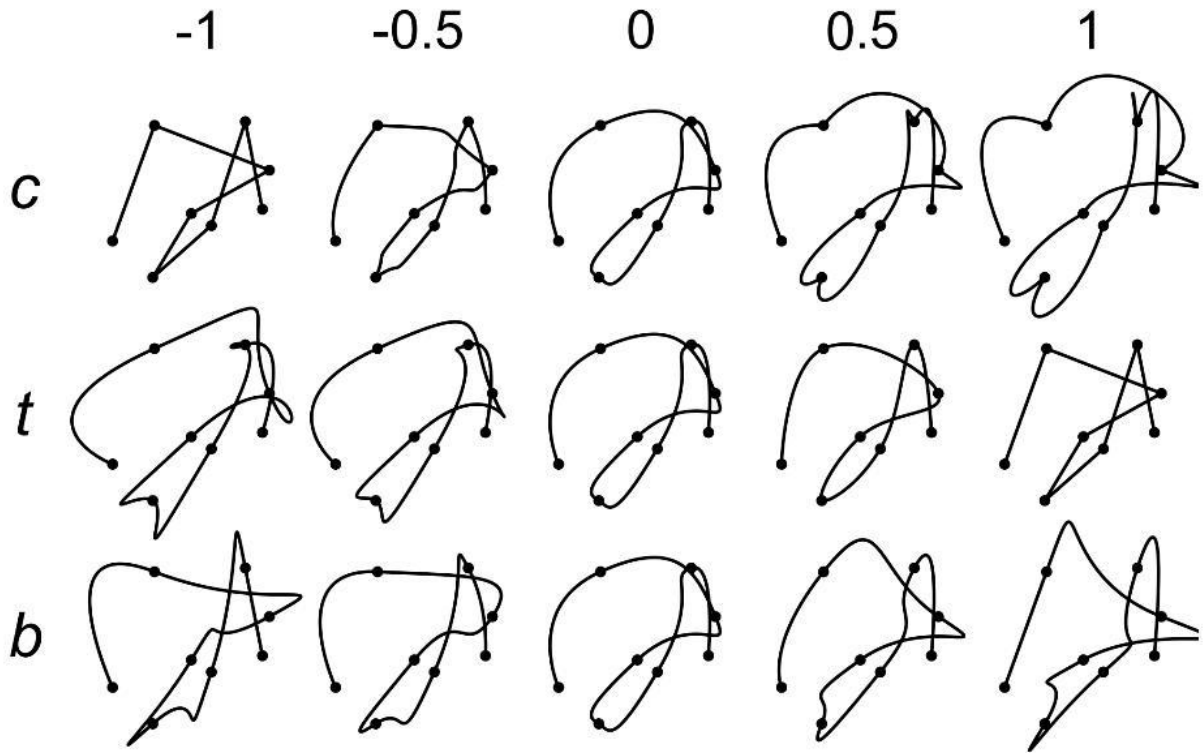


Fig. 39 Visualization of the KB Spline for different values of tension, bias and continuity [14].

The effects of individual parameters is summarized in Table 3 and shown in Fig. 39 [14]:

Table 3 Effects of different parameter values on the KB spline.

Parameter/Value	1	-1
Tension	Tight	Round
Bias	Post shoot	Pre shoot
Continuity	Inverted corners	Box corners

5.3.3 Output Files Generation

The export of the animation data files for the PBRT rendering requires generating a vast amount of data. Already in previous work, the MPII data model was divided into separate layers so that working with the model could be made easier. Loading the whole model into memory was nearly impossible on common computer stations. Similar care has to be taken when generating the input files for rendering of animations. The software application should implement tools that will enable the user to generate the input data with as least of redundant work as possible.

A great advantage of VRML and PBRT format is that they both support storing the model in separate files and loading it together through “includes”. This is especially handy for generating the animation files. Most of the scene geometry will not be affected by the animations at all. Only MPII layers containing doors, windows, and window shutters can undergo geometry modifications. Animation of the camera only requires inserting new camera specification into the PBRT file. The same is valid for the light sources.

Hence, the implemented software application should enable combining animated parts of the scene with the static parts, so that the input data can be prepared as conveniently as possible.

5.4 Additional Extensions

A need for additional extensions to the previously implemented application originated during the development process. The extensions described and analyzed in this section are not directly related to the scope of this thesis but more to the usability of the implemented work. The two major requirements on the new version of the developed software application were as follows:

- Implementation of project status, which could save the sequence of actions performed by the user on particular data set. The project status could then be stored and consequently loaded into the application. The project status would substantially reduce the redundancy in the work of the user. Specific geometry modifications and animation processing could be logged into the project and then loaded in the future, without any need to repeat the previous work.
- Developing a console version of the VRMLtoPBRT application. This console version should be free of the wxWidgets library and only the standard input/output interface for data processing. In this manner, the possible future compatibility issues of the application should be significantly reduced.

5.4.1 Project Status

The project module should be composed of three main components. The first component should keep track of any actions performed by the user. The second component should be able to save the project status into a text file and load it back into the application from a given text file. The final third component should be able to execute the loaded project status and apply the desired actions to the specified input data.

Keeping track of the project status will require implementing an action node associated with each possible action of the user. The log of these actions will be a linked list connecting all the action nodes.

Storing the project status into a text file can be implemented as simple as traversing the list of action nodes and converting them into a suitable textual representation, which would be written into the output file. In a similar manner, loading the project status from a text file comprises mainly initializing an empty action list and the inserting action nodes according to their textual description in the input text file.

Finally, the execution of project status is composed of traversing the list of action nodes and executing appropriate actions for every single node. Different action nodes will be needed. Possible types of action nodes are as follows:

- Action nodes for loading the input data.
- Action nodes for specifying the renderer settings.
- Action nodes for storing the parameters of the global animation object.
- Action nodes implementing individual geometry modifications for specific objects of interest.

- Action nodes determining the output actions performed with the modified data model.

5.4.2 Console Version of VRMLtoPBRT Application

Despite of its modular architecture, the previously implemented VRMLtoPBRT application is extensively interlaced with the wxWidgets library. Not only is this library used for implementing the graphical input/output interface, but also wxWidgets input/output data streams and some data types are used. As the MPII data model and its auxiliary software application are intended for long term use by the computer graphics community, this interlacing with the wxWidgets library might bring up some considerable compatibility issues in the future. For this reason, it is reasonable to develop a console version of the VRMLtoPBRT application, which would be free of the wxWidgets library. This application would only use standard input/output interface and standard data types.

Implementation of the console version will take a great use of the implemented project status. It is intended that the VRMLtoPBRT application will take a project text file as an input and process the data accordingly. Specific geometry modifications and other actions will need to be defined inside the project status files.

6 Implementation

This chapter discusses the implementation of the solution design proposed in Chapter 5. First, the extensions to the architecture of the previously implemented application are listed and explained. Next, the description of the data model modifications follows. The MPII data model had to be substantially modified by redoing the model hierarchies of the objects of interests and renaming the object accordingly. The discussion of the implemented scene representation is presented afterwards. Here, the new data structures for storing the objects of interest and working with them are explained. Further, the implementation of animation lists, animation nodes and their application to the objects is described. Finally, the GUI extensions and additional implemented functionalities are explained.

6.1 Application Architecture

In order to maintain consistency with previous work [1] the implemented architecture is presented as a set of extensions to the earlier work. As discussed earlier, the new functionalities of the application did not require an addition of new modules and interfaces. Instead, the already existing modules had to be extended. A diagram of the implemented architecture is presented in Fig. 40. In order to distinguish the contribution of the presented work from the previously existing parts, the new extensions are highlighted by red color.

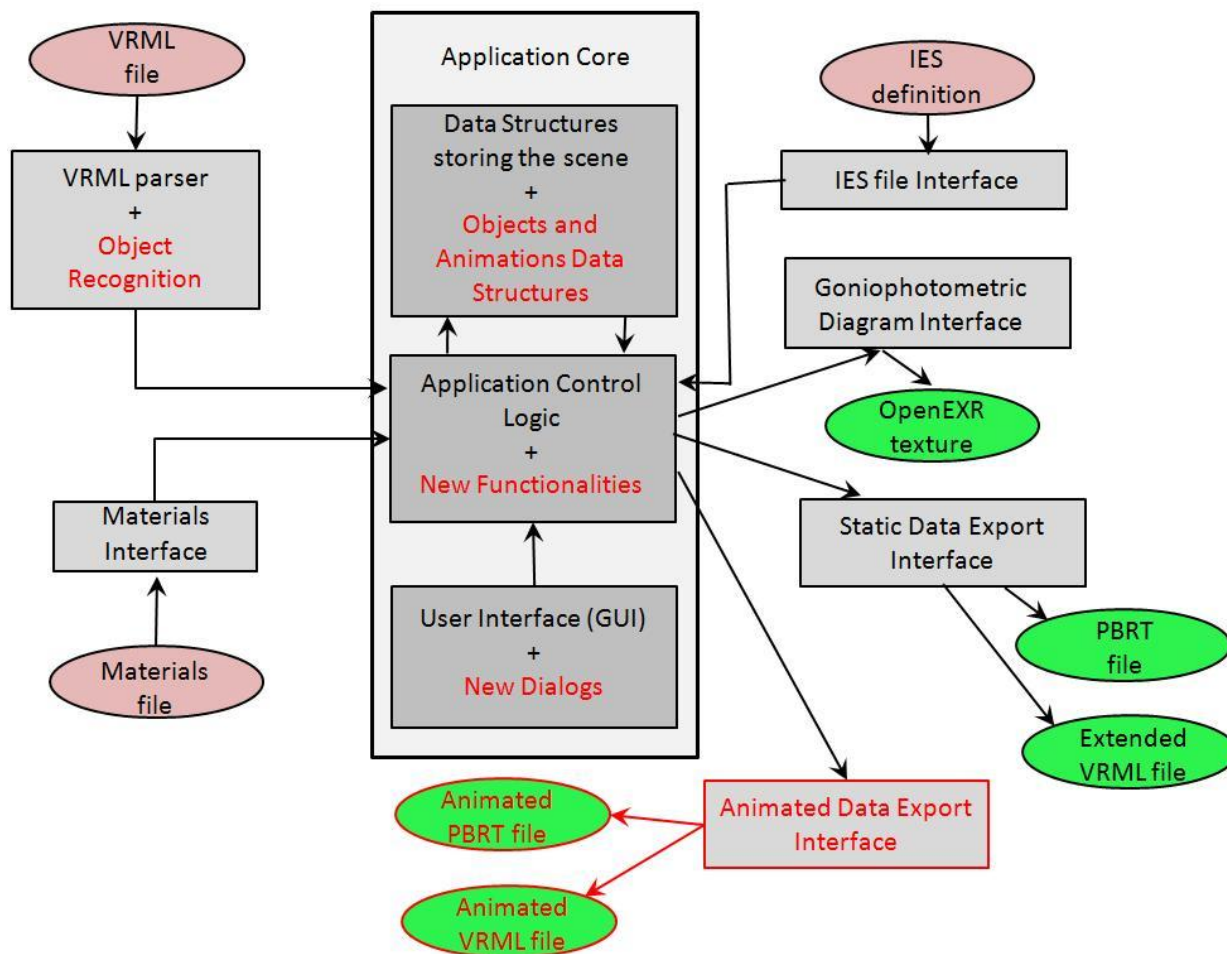


Fig. 40 Implementation of the data conversion application.

The new application works with the updated MPII model, which now contains reordered object hierarchies with accordingly labeled nodes. The implemented extensions are as follows:

- The interface for parsing the input VRML file was extended to recognize the objects of interest (doors, windows, window shutters) and to initialize appropriate data structures.
- New data structures were implemented to store the information about the objects of interest (name, geometrical modifications, animations, etc.).
- The GUI of the application was extended by several new dialogs for displaying, modifying and exporting the new data model.
- New functions and data flow connections were added to the application control logic in order to connect the user interface with the internally stored data structures.
- The PBRT and VRML data export functions were modified to account for the geometry modification that the user can insert into the model.
- A new interface for exporting the animations into the PBRT and VRML format was added. This includes maintaining an internal time generator, which synchronizes the animation of all objects and cameras. Also several new functions were implemented for convenient processing and maintenance of the animation data files.

Detailed description of the implementation of all introduced extensions follows.

6.2 Data Model

The first necessary step was to update the MPII data model. The modifications to the model were threefold. First, the object hierarchies had to be appropriately remodeled and renamed. This was needed for the application to recognize the objects and their important parts. Second, lights were installed inside rooms and offices. In the previous static MPII model all doors were closed, there was no access to individual rooms and offices in the building. This had to be fixed in order to maintain accurate lighting conditions after doors are opened. Finally, it was discovered that the VRML export module of the 3DS Max cannot handle mirror transformation. These resulted in incorrect geometry conversion into the VRML file. This issue had to be corrected by removing the mirror transformations from the model.

6.2.1 Object Hierarchies

The previously implemented VRML parsers only recognized the standard nodes of VRML. Thus, the lights could be easily identified, because they were exported as the *SpotLight* node in the VRML format. This was not the case for all the objects of interest as they were all exported as transform nodes. The implemented solution was to construct object hierarchies in a predefined specific format by using group nodes in 3DS Max. By introducing a consistent naming convention for all nodes of the hierarchy, the implemented parser could be extended to identify VRML transform nodes containing certain objects of interest.

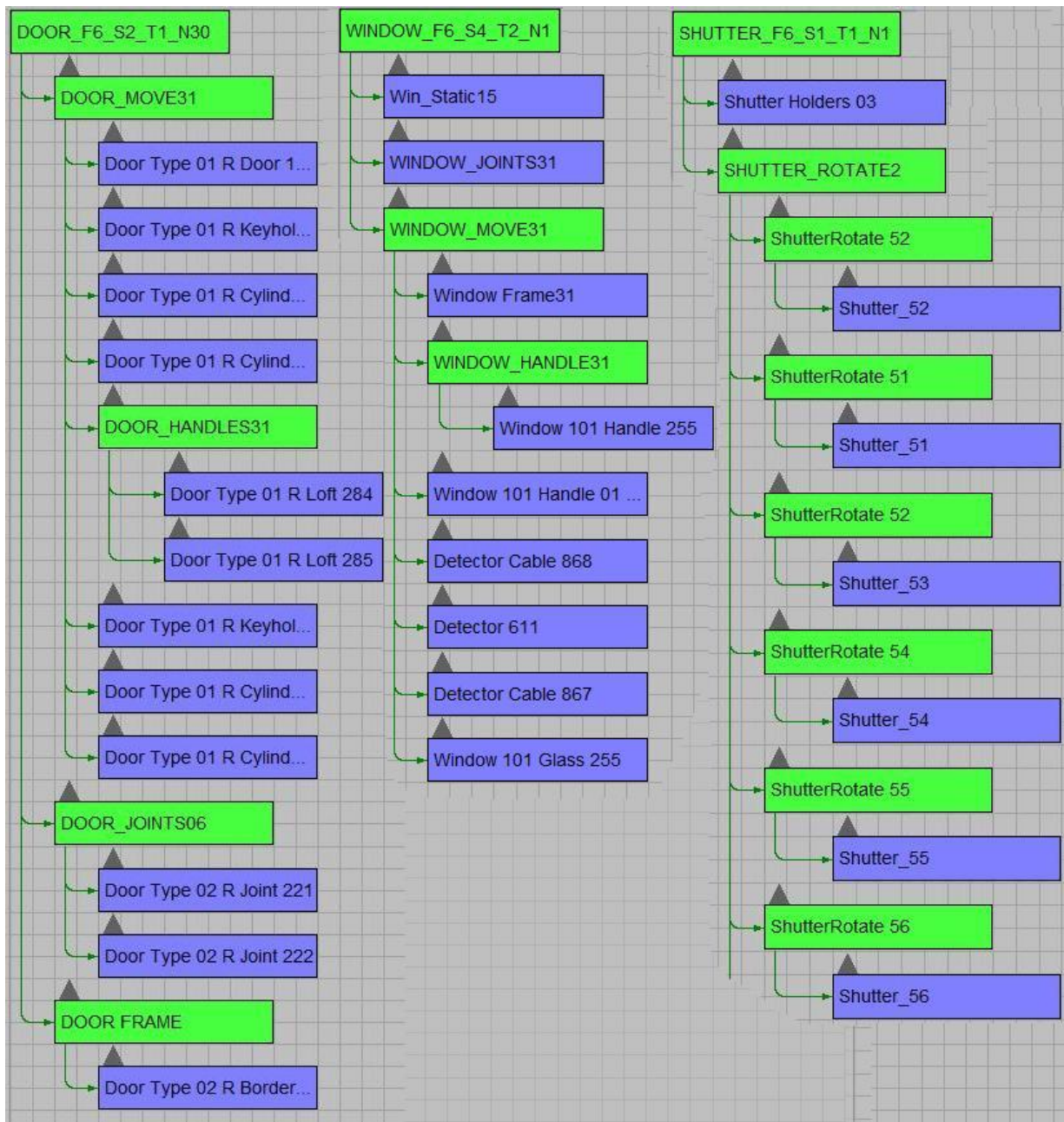


Fig. 41 Examples of the modified MPII 3DS Max model. From left to right are exemplary object hierarchies for door, window and window shutter objects.

The actual implementation required remodeling every single door, window and window shutter object that was to be made available to the user for geometry modifications and animations. All of these objects had to be labeled according to the object naming convention as proposed in Section 5.2.2. Fortunately, the 3DS Max offers user friendly tools for object instancing and copying. A newly remodeled object could be copied multiple times. Even more importantly, up to minor differences the 2nd through the 6th floor are almost identical. Therefore, after completing one floor the whole layer could be copied to the floor below, which accelerated and simplified the work substantially.

Examples of the object hierarchies as they appear in 3DS Max are shown in Fig. 41. The objects hierarchies as designed in Section 5.2.1 can be recognized in the figure. Furthermore, notice the object naming convention in the root nodes, which allows for location and type identification of the object. Table 4 lists the number of particular objects remodeled in the MPII model.

Table 4 Number of remodelled objects of interest.

Object Name	Number of Objects
Doors	315
Windows	230
Window Shutters	187

6.2.2 Adding New Lights

The original MPII model constructed by Josef Zajac was previously improved by adding lights with measured luminaries. However, since all the cameras in the static model allowed access only to the main atrium and the connected halls, no lights were placed in individual rooms and offices. In the current work, the doors in the MPII building can be opened and the rooms can be explored upon user's desire. In order to maintain accurate lighting conditions in the scene, the lights in the offices had to be inserted into the model.

Unfortunately, very little documentation on the types and locations of room lights could be found. The leading source was a scene modeled by Josef Zajac, which shows one of the offices with the Radium-type light placed exactly in the middle of the room. Another view of the lighting in another office was provided by Dr. Vlastimil Havran in a form of a video sequence. This video showed four Radium lights placed in two rows in the larger rooms. Using this partial information, the lights were added into particular office in the 2nd through the 6th floors. In this manner, 350 new lights were added into the model. However, it is important to finish the model by documenting the actual positions and types of lights in the real MPII building and updating the MPII data model accordingly.

Comparison of the MPII model before and after the addition of new lights can be seen in Fig. 42. The figure shows a top view of the 6th floor of the MPII building, where lights are denoted by a crossed circle. In a similar manner, lights have been added into offices in the 2nd – the 5th floor.

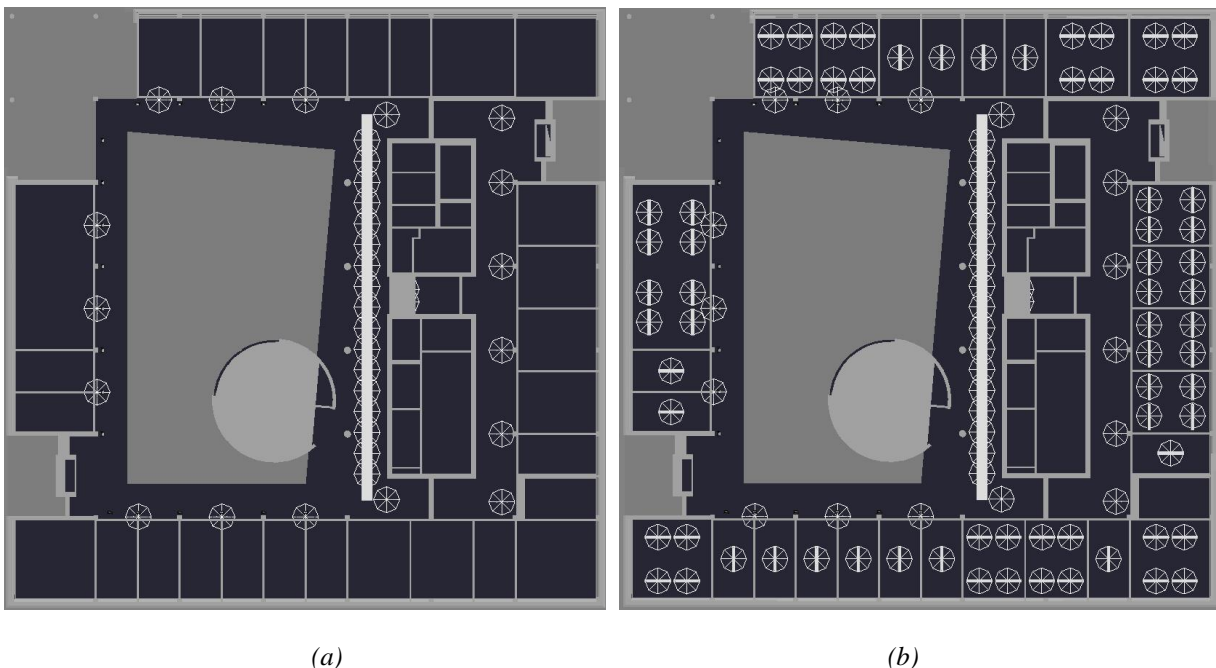


Fig. 42 The 6th floor in the MPII building without (a) and with (b) the added room lights.

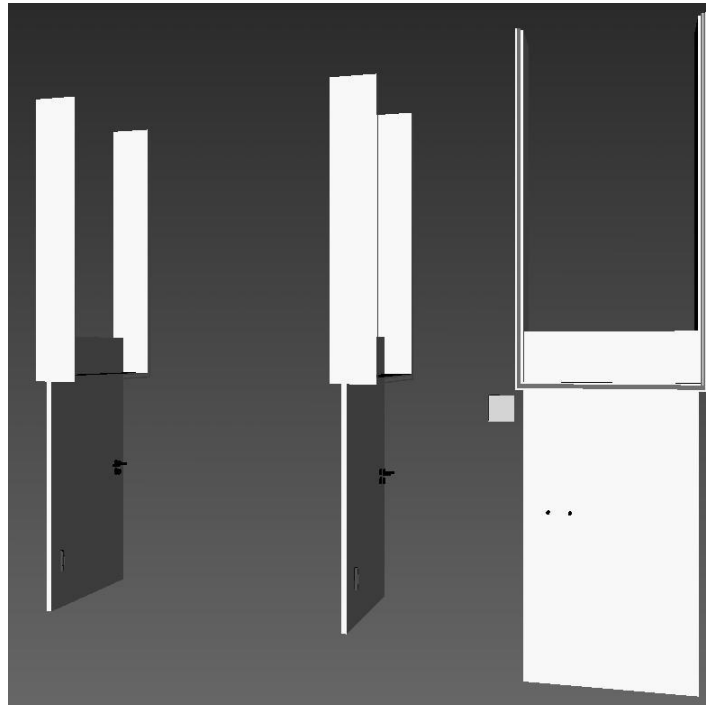


Fig. 43 Incorrectly transformed doors in VRML.

6.2.3 VRML Errors Corrections

Initially, a visual inspection of all 3DS Max models and VRML scenes provided from previous work was performed. This inspection revealed substantial inconsistencies between the two models. For instance, multiple doors were transformed incorrectly into VRML with some parts upside down or incorrectly rotated. This inconsistency is illustrated in Fig. 43.

A literature research revealed that the mirror transformations applied during object modeling in 3DS Max are the most likely cause of these problems. This transformation creates a copy of specified object by mirroring it around its central axis. Mathematically, this transformation is performed as a scaling operation with negative coefficients. In the same manner, this transformation is also exported into the VRML format. However, this is in conflict with the VRML standard [13]. The standard clearly defines that the scaling coefficients must have values greater than zero. Otherwise the behavior of the model is undefined. In case of the used VRML viewer – Cortona 3D Viewer 6.0 [21], this resulted in incorrect models as shown in Fig. 43.

In order to ensure correct visualization of the model, the mirror transformations had to be removed from the MPII model. A simple procedure was used to achieve this task. Model of each incorrectly mirrored object, was saved into a separate file and its original was deleted from the model. Consequently the separated object was merged again with the original model. By separating the object, the connections and instancing in the model were disrupted. After merging, the exact geometry was inserted into the original model and the mirror transformation was omitted.

6.3 Scene Representation

The internal data structures storing the loaded geometry of the scene have been already implemented in previous work. A detailed overview of this implementation can be found in [1]. However, in order to enable for geometry modifications and animations of the MPII

model additional data structures and functions for recognizing the objects of interest in the VRML input file had to be implemented.

6.3.1 Data Structures

Each of the objects of interests (doors, windows, window shutters) are stored as an ordinary VRML *Transform* node. By parsing the name of the *Transform* node (described later in Section 6.3.2), nodes which contain the objects of interest can be identified. The previous code was extended by adding the type of object and a pointer to the information about the object to each *Transform* node. This extension can be seen in the following code snipe:

```
class NodeTransform : public Node
{
    ...
    /// Type of the transform node object
    ObjectType objectType;
    /// Pointer to the objectInfo object, if this node is not an
    /// recognized object, this pointer is set to NULL
    ObjectInfo * objectInfo;
    ...
}
```

Because some properties of the objects of interest are common to all of them (e.g. floor number, section number, name), a common parent class was implemented – *ObjectInfo*. Individual objects then inherit from this parent class and add attributes unique for each object. Description of these classes is presented bellow.

ObjectInfo Class

The *ObjectInfo* class serves as a parent class to the *DoorInfo*, *WindowInfo* and *ShutterInfo* classes. It stores the common parameters, such as name, floor number, section number and the construction type of the object. The construction type is a very important property, since it allows for distinguishing among various types of doors. Different types of doors require different transformations to be applied in order to open them. Furthermore, the *ObjectInfo* class stores information about the object havening been animated and also its animation list (explained below in Section 6.4.1). A fragment of the class declaration is shown below.

```
class ObjectInfo{
    /// Name of the object node
    char* name;
    /// Floor where the object is located
    ObjectFloor objectFloor;
    /// Section where the object is located
    ObjectSection objectSection;
    /// Type of the info object
    ObjectInfoType objectInfoType;
    /// Construction type of particular object
    ObjectConstructType objectConstructType;
    /// Has the object geometry been modified
    bool mHasModified;
    /// Flag determining if the object has been animated or not
    bool mHasAnimated;
    /// Pointer to a list with animation checkpoints
    AnimationList * animList;
public:
    // Function defintions
}
```

```

    ...
}

```

***DoorInfo* Class**

The *DoorInfo* class inherits from its parent *ObjectInfo* class. It adds three important attributes related to the geometrical modifications of the door object. The opening rate of the door and the door handle position are stored along with the axis for door opening. This axis is determined based on the position of door joints. The opening rates are stored as integer numbers denoting the percentage ratio of opening of particular component. The range of values is between 0 and 100%. The following code snippet shows the declaration of the class:

```

class DoorInfo : public ObjectInfo {
    /// The amount of rotation
    int mRotateDeg;
    /// The axis of the rotation (the coordinates of the joints)
    Vector3D axis;
    /// The amount of handle opening
    int mHandleRate;
public:
    // Function definitions
    ...
}

```

***WindowInfo* Class**

The *WindowInfo* class extends the parent *ObjectInfo* class in a similar manner as the *DoorInfo* class. The window open rate, window handle rotation rate and the window axis are stored here. The opening rates are stored as integer numbers denoting the percentage ratio of opening of particular component. The range of values is between 0 and 100%. In addition, the object maintains the type opening applied to the window. It could be either opened in horizontal or vertical direction. Below is a part of the *WindowInfo* class declaration.

```

class WindowInfo : public ObjectInfo{
    /// The amount of rotation
    int mRotateDeg;
    /// The amount of handle opening
    int mHandleRate;
    /// The axis of the rotation (the coordinates of the window joints)
    Vector3D axis;
    /// type of the opening - horizontal - 0, vertical - 1
    int mOpenType;
public:
    // Function definitions
    ...
}

```

***ShutterInfo* Class**

Finally, the *ShutterInfo* class stores attributes unique for the window shutter object. It extends the parent *ObjectInfo* class by storing the scaling rate (open rate), rotation rate and the top offset coordinate of the shutter object. The opening rates are stored as integer numbers denoting the percentage ratio of opening of particular component. The range of values is between 0 and 100%. The top offset coordinate is used for translating the shutter object after it has been modified by the scaling transformation. The following code snippet shows the *ShutterInfo* class declaration.

```

class ShutterInfo : public ObjectInfo {
    /// The amount of scaling
    int mScaleRate;
    /// Y translation offset of the top of the shutter
    float mTopOffset;
    /// The shutter rotate opening rate
    int mRotateRate;
public:
    /// Function definitions
    ...
}

```

6.3.2 Object Recognition

New functions for recognizing the objects of interest in the input VRML file had to be implemented. These functions are grouped in *geometryModify.h* and *geometryModify.cpp* files. The recognition of individual objects is based on specific key words that are searched for in the names of the identified VRML *Transform* nodes. The procedure for object recognition is based on the following scheme:

1. Use the VRML lexical and syntactical analyzers to parse the VRML input file. Initialize the internal data structures according to the scene geometry (previously implemented).
2. Iterate through all *Transform* nodes. Find all *Transform* nodes that contain the specific key words.
3. Initialize the *ObjectInfo* objects based on the matched key word.

All of the implemented key words are defined in the *constants.h* file. As an example of this procedure, consider function *CheckListDoors()*, which initializes all door objects. It takes list of all identified VRML nodes as an input and it returns the number of found door objects.

```

int CheckListDoors(vvector<Node*>& cont)
{
    int numbDoors = 0;

    for(int i = 0; i < cont.size(); i++){
        /// The current node
        Node* temp = cont[i];

        /// Find the group node of the whole floor
        if(CheckNodeNameCode(temp->retName(), CODE_DOOR_FLOOR)){

            NodeTransform* tempParT = static_cast<NodeTransform*> (temp);

            for (int j = 0; j < tempParT->retChildNum(); j++){
                /// We are only interested in nodes that contain door
                /// objects
                Node * tempCh = tempParT->retChildIdx(j);

                if(CheckNodeNameCode(tempCh->retName(), CODE_DOOR)){
                    /// Retype to a Transform node
                    NodeTransform* tempT = static_cast<NodeTransform*>(tempCh);
                    numbDoors++;

                    /// Sets the correct type
                    tempT->setObjectType(_TYPE_DOOR);
                }
            }
        }
    }
}

```



```

        if (tempT->retName() != NULL) {
            tempT->initDoorInfo();
            tempT->retObjectInfo()->initAnimList();
            tempT->retObjectInfo()->correctObjectName(tempT->retName());
        }
    }
}
}
}
// Return the number identified door objects
return numbDoors;
}

```

The function iterates through all of the nodes in the provided list. It uses the *CheckNodeNameCode()* function to search for a specific key word in the name of the node. When a group node that stores all the door objects is found, all its children are searched through. After a door object is identified, the type of the *Transform* node is set appropriately. Next the *ObjectInfo* pointer is initialized as a *DoorInfo* object. Finally, the *correctObjectName()* function is called. This function parses the name of the *Transform* node (written in a specific format as described in Section 5.2.2) and it extracts the important attributes such as floor number, section number, identification number and the object construction type. Finally, after all nodes are analyzed, the final number of identified door objects is returned.

Slightly modified functions are used for identifying all window and window shutter objects.

6.4 Animations

This section describes the implementation of scene animations. It presents the implemented data structures, important animation functionalities as well as a discussion of animation post-processing.

6.4.1 Data Structures

Several new data structures had to be implemented and incorporated with the existing architecture in order for the animation processing to be possible. These data structures work at different levels of the program data flow. First, a global animation object was implemented. This object stores the global animation settings and it works as a global timer, synchronizing the animations of individual objects. Secondly, each animation-enabled object maintains an animation list. Every animation list stores a sequence of animation nodes. These animation nodes are unique to different types of objects of interest. Explanation of implemented data structures follows.

Global Animation Object

The global animation class – *AnimationInfo*, stores the global properties of the animation sequence. In addition, it also works as a global timer. The user can set the length of the animation in seconds (*mAnimLength*) as well its frame rate (*mAnimFps*). These values are then used for determining how many individual frames should be generated. Furthermore, global settings such as flags determining automatic door and window handles opening or the global parameters of the KB spline (described in Section 5.3.2) for interpolation are stored here.

The current time (*mCurrentTime*) and frame number (*mCurrentFrame*) are both encapsulated in the *AnimationInfo* class. At the start of the animation processing, the initial time is set by calling the *setCurrentTime()* function. After preparing data for each frame, function *incCurrentTime()* increases the current time and the frame number. Code snippet describing the implementation of the *AnimationInfo* class follows:

```
class AnimationInfo{
    /// Length of the animation [seconds]
    float mAnimLength;
    /// Frame rate of the animation [frames per second]
    float mAnimFps;
    /// Has the animation been set
    bool mHasBeenSet;
    /// Additional flag determining if we are performing the animation
    ///during the output
    bool mDoAnimate;
    /// The current time during the animation [seconds]
    float mCurrentTime;
    /// The current frame number during the animation
    int mCurrentFrame;
    /// Time step (1/fps)
    float mTimeStep;
    /// Flag determining if the handles should be animated automatically
    bool mHandlesAuto;
    /// The duration of handle animation [frames]
    int mHandlesDuration;
    /// Global parameters for adjusting the KB Splines for camera
    ///interpolation
    /// Tension
    float tension;
    /// Continuity
    float continuity;
    /// Bias
    float bias;
    ...
public:
    /// Function definitions
    ...
    /// Sets the current time of the animation
    void setCurrentTime(float time) { mCurrentTime = time; }
    /// Increases the current time
    void incCurrentTime(void) {mCurrentTime += mTimeStep;}
    ...
}
```

Animation List

The animation list constitutes a key component for the animation processing. It stores a sequence of events that drives the animation process. Each event specifies a state of the animated object at particular time. The state is composed of specific values of all dynamic parameters of the object. A pointer to the animation list is stored in each *ObjectInfo* object.

The animation can be dynamically created by the user. Events can be added into the list in an unordered sequence. Further, events can be deleted upon user's request. The actual implementation of the *AnimationList* class automatically keeps the list of events in a sequence ordered according to time. It is implemented as a doubly connected linked list of animation nodes. The class only stores pointer to the head and to the tail of the linked list. In addition the specification of the interpolation mode to be used is maintained here. New animation nodes can be added by calling the *insertNode()* function and deleted by providing the node index to

the *deleteNode()* function. Below is a code snippet demonstrating the implementation of the *AnimationList* object.

```
class AnimationList{
    /// head of the list
    AnimationNode * head;
    /// Pointer to the last node in the list
    AnimationNode * last;
    /// Mode of the interpolation used during this animation
    /// 0 - linear, 1 - cosine, 2 - cubic, 3- KB Splines
    int interpMode;
    /// Number of nodes in the list
    int nodeCount;
public:
    ...
    /// Inserts node into the list - nodes are automatically ordered
    /// according to the time
    /// @return False if the node cannot be inserted (node at this time
    /// already exists in the list)
    bool insertNode(AnimationNode * node);
    /// Deletes node with particular index number
    /// @return True if the deletion went fine, false otherwise
    bool deleteNode(int index);
    ...
}
```

The animation list works with the *AnimationNode* object, which is a parent class for all of the specific animation nodes.

***AnimationNode* class**

Similarly to the *ObjectInfo* class, some properties of the animation nodes are common to all objects. Every animation node must store the time of the animation event (*animTime*). This animation time denotes the frame number when the event occurs. Also, since the animation nodes form together a doubly connected linked list, each animation node stores pointers to the next (*animNext*) and to the previous (*animPrev*) animation node in the ordered sequence.

These general attributes were encapsulated in the *AnimationNode* class, which serves as a parent class to the more specific animation nodes for each type of object. This implementation enabled maintaining a single type of general animation list, which only works with the *AnimationNode* class and abstracts from the actual object being animated. The following code snippet shows the implementation of the *AnimationNode* class.

```
class AnimationNode{
    /// Time of the animation [frame]
    int animTime;
    /// Flag determining if this is an automatic node
    bool autoNode;
    /// pointer to the next Animation node in the list
    AnimationNode * animNext;
    /// pointer to the previous Animation node in the list
    AnimationNode * animPrev;
public:
    /// Functions definitions
    ...
}
```

***AnimationNodeDoor* class**

The door animation object extends its parent *AnimationNode* by adding the parameters specific for animation of doors. It specifies the opening rate of the door and the rotation of the door handle. The rates are stored as integer numbers denoting the percentage ratio of opening of particular component. The range of values is between 0 and 100%.The following code snippet shows the implementation of this class.

```
class AnimationNodeDoor : public AnimationNode{
    /// Opening rate of the door
    int doorOpen;
    /// Opening rate of the door handle
    int doorHandle;
public:
    // Functions definitions
    ...
}
```

***AnimationNodeWindow* class**

The window animation node also adds the opening rate of the window and the rotation rate of the window handle. The rates are stored as integer numbers denoting the percentage ratio of opening of particular component. The range of values is between 0 and 100%. Additionally, it adds the type of the opening mode, which can be either vertical or horizontal.

```
class AnimationNodeWindow : public AnimationNode{
    /// Opening rate of the window
    int windowOpen;
    /// Opening rate of the window handle
    int windowHandle;
    /// Opening type of the window
    int windowOpenType;
public:
    // Function definitions
    ...
}
```

***AnimationNodeShutter* class**

The shutter animation node stores the values of the open rate for the window shutter object and the rotation of individual window shutter layers. The opening rates are stored as integer numbers denoting the percentage ratio of opening of particular component. The range of values is between 0 and 100%.

```
class AnimationNodeShutter : public AnimationNode{
    /// Opening rate of the shutter
    int shutterOpen;
    /// Rotation rate of the shutter
    int shutterRotate;
public:
    // Function definitions
    ...
}
```

***AnimationNodeCamera* class**

The most important attributes stored in the camera animation nodes are the position and the orientation of the camera. The orientation is stored as the up vector of the camera and a rotation around this direction. Furthermore, the KB spline interpolation method can be controlled by three locally varying parameters – tension, continuity and bias. Specifying their values for every single animation node allows for robust control of the interpolation method.

```
class AnimationNodeCamera : public AnimationNode{
    /// Position of the camera
    Vector3D cameraPosition;
    /// Orientation of the camera
    Vector4D cameraOrientation;
    /// Parameters for the local adjustment of the KB Spline
    /// Tension
    float tension;
    /// Continuity
    float continuity;
    /// Bias
    float bias;

public:
    /// Function definitions
    ...
}
```

***AnimationNodeLight* class**

The final type of animation node is the light animation node. The only two attributes stored here are the on/off status of the light and its current intensity.

```
class AnimationNodeLight : public AnimationNode{
    /// Is the light on/off
    bool lightOn;
    /// The intensity of the light
    int lightIntensity;
public:
    /// Function definitions
    ...
}
```

6.4.2 Animation Specification

Despite of the GUI not providing a typical graphical interface for creating an animation sequence (such is the case of the 3DS Max), an effort was undertaken to create a user friendly way for defining the animation sequences. The user can specify the animation sequence by following a simple multi-step procedure:

1. Select the object of interest to be animated (e.g. door object).
2. Enable animations of this object.
3. Specify the parameter values at certain time event.
4. Add the animation node to the animation list.

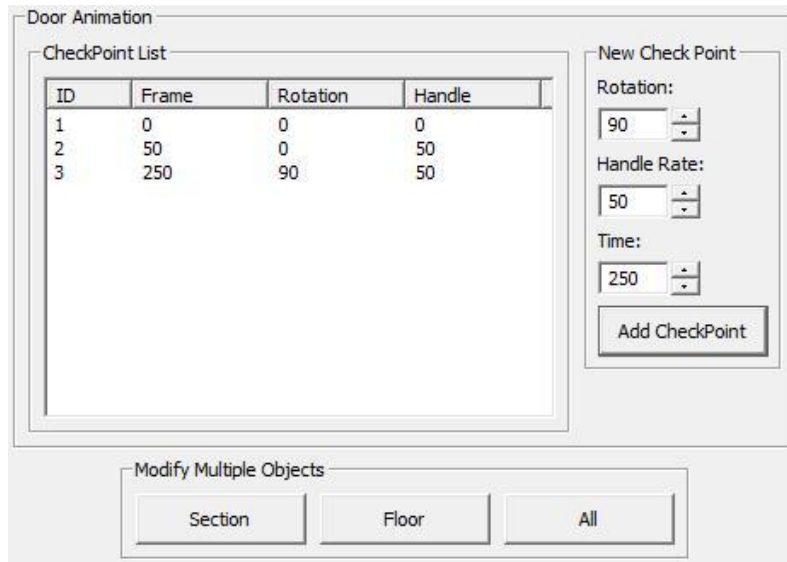


Fig. 44 Animation specification.

5. Select the interpolation method for this animation.
6. Select the application mode for this animation.

Prior to starting with the animation definition, the global animation object has to be initialized with the proper values. By default, the time of animation is set to zero seconds, thus disabling the animation interface. After specifying a certain non-zero length the user can proceed with the animation specification.

The animation nodes can be added to the animation sequence (inserted into the animation list) in a time-unordered chronology. The animation list automatically orders individual animation nodes according to the specified time. It also checks for redundancies, since the object of interest can be only in a single unique state at certain time event. Nodes with time event that is already present in the animation list are not added into the list. In addition, animation nodes can be easily deleted from the animation list.

In many situations it is desirable that multiple objects will undergo the same animation sequence. For example consider the following scenario. The user would like to turn off all lights on the 2nd floor at certain time of the animation. Instead of specifying a separate animation sequence for every single light object, an application mode can be selected for the animation. After the animation sequence is defined, the user can assign this animation list either to the currently selected individual object, to all objects on the same floor, to all objects on the same floor and in the same section, or to all objects in the whole building.

An example of animation definition using the GUI of the developed application is given in Fig. 44. Here, an animation of a door object was created. The animation list contains three animation nodes. First node was placed at the 0th frame and defines the initial state of the object – closed door and closed door handle. The following animation node was placed at the 50th frame and it specifies that the door handle is now opened at 50% of its rotation. The last animation node was placed at the 250th frame and it defines that the door is opened at 90% of its rotation. On the right side of the figure the interface for adding additional animation nodes can be seen. Below, the application mode of this animation sequence can be chosen.

6.4.3 Interpolation

A specific interpolation function (see 5.3.2) can be selected by the user for each animation sequence. This function defines how parameter values are interpolated between individual animation nodes. Each children of the parent *ObjectInfo* class can automatically calculate its current state. For example the current door opening and the current rotation of the door handles can be obtained by calling the `DoorInfo::retCurrentOpening(void)` and the `DoorInfo::retCurrentHandleRate(void)` functions. These functions first obtain the current animation time from the global animation object. Next the currently active two neighboring animation nodes are searched for in the animation list. Finally, the interpolated value is obtained by using the specified function to interpolate between the values of the two animation nodes. The following code snippet shows part of the implementation of the `retCurrentOpening(void)` function.

```
float DoorInfo::retCurrentOpening()
{
    ...
    // 1) First find the two surrounding animation checkpoints
    AnimationNode * help = this->retAnimationList()->retListHead();

    while (animInfo.retCurrentFrame() > help->retAnimTime()) {
        help = help->retAnimNext();
    }

    // Check if we are at the very beginning of the animation
    if (help == this->retAnimationList()->retListHead()) {
        AnimationNodeDoor* tempDCurr = static_cast<AnimationNodeDoor*>
            (help);
        return tempDCurr->retDoorOpen();
    }

    AnimationNodeDoor* tempDPrev = static_cast<AnimationNodeDoor*>
        (help->retAnimPrev());
    AnimationNodeDoor* tempDCurr = static_cast<AnimationNodeDoor*>
        (help);

    // 2) Calculate the time fraction between the identified neighboring
    // checkpoints
    float frameRel = (animInfo.retCurrentFrame() -
        tempDPrev->retAnimTime()) / (float)(tempDCurr->retAnimTime() -
        tempDPrev->retAnimTime());

    // 3) Based on the current interpolation mode, calculate the opening
    // rate based on the time fraction
    float openRate = 0;

    // Linear interpolation
    if (this->retAnimationList()->retInterpMode() == 0) {
        openRate = linearInterp(tempDPrev->retDoorOpen(),
            tempDCurr->retDoorOpen(), frameRel);
    }

    // Cosine interpolation
    else if (this->retAnimationList()->retInterpMode() == 1) {
        openRate = cosineInterp(tempDPrev->retDoorOpen(),
            tempDCurr->retDoorOpen(), frameRel);
    }

    // Cubic interpolation
```

```

else {
    float v1 = tempDPrev->retDoorOpen();
    float v2 = tempDCurr->retDoorOpen();
    float v0 = v1;

    if (tempDPrev->retAnimPrev() != NULL) {
        AnimationNodeDoor* tempDPrevPrev =
            static_cast<AnimationNodeDoor*>
            (tempDPrev->retAnimPrev());

        v0 = tempDPrevPrev->retDoorOpen();
    }

    float v3 = v2;

    if (tempDCurr->retAnimNext() != NULL) {
        AnimationNodeDoor* tempDCurrCurr =
            static_cast<AnimationNodeDoor*>
            (tempDCurr->retAnimNext());

        v0 = tempDCurrCurr->retDoorOpen();
    }

    openRate = cubicInterp(v0, v1, v2, v3, frameRel);
}
return openRate;
}

```

It can be seen in the code how the two neighboring animation nodes, *tempDPrev* and *tempDCurr*, are located. Next, the time fraction between the two animation nodes is calculated using the current time from the global animation object - *frameRel*. Finally, the resulting door open rate – *openRate* – is calculated using the specified interpolation method.

In analogical manner, the interpolation of other objects of interest is handled.

6.4.4 Animation File Processing

The previous section described the data structures for storing the animation sequences. Also the animation definition procedure was explained. This section describes supports and functionalities that are needed for generating correct output animation data for the PBRT rendering engine. Since, the animation processing can involve producing a large amount of output data, it is vital to eliminate redundant work and make the animation processing as convenient for the user as possible. In the following text the implemented animation processing functionalities are described.

Create Individual Animation Files

The most straightforward method for generating the output animation data is producing a single PBRT file for every individual frame. Here, the renderer specification together with the camera, lights definitions and the geometry of the scene are stored in a single file. The names of the files are appropriately indexed based on the current frame number. This functionality is also implemented for the VRML animation file processing.

Create Separated Animation Files

Due to its complexity the MPII data model was decomposed into multiple layers. Therefore the creation of the final animation input data for the PBRT rendering engine will

inevitably involve combining together several distinct layer of the model. In order to make this a user-friendly the separated animation files functionality can be used. Here, a main file is created for every individual frame. This main file stores the renderer definitions, camera and lights specifications, but not the geometry data. The geometry data for each layer (either animated or static) are stored in a separate geometry file. The geometry file is then included at the end of the main file. In this manner, multiple layers of the MPII model can be included into a single main file. These layers will then be combined by the PBRT rendering engine into a single complex scene. This functionality is also supported for the VRML animation file processing.

Add Static Geometry

Suppose that the user is creating an animation composed of a building geometry and animated doors. The door animation requires generating new geometry file for every frame. However, the building geometry stays static during the whole animation. Therefore after main and geometry files were generated for the animated doors, a single static geometry file for the building construction needs to be included into every main file. This operation does not require generation of any input data, only the include directive with the appropriate file name has to be added into the main files. This functionality is also supported for the VRML animation file processing.

Add Animated Geometry

In certain situations the user might have already generated the animated geometry files for certain layer of the model (e.g. doors). If this geometry data are to be reused in another animation sequence, it is unnecessary to generate the output data again. Instead, only the correctly indexed file names should be added together with the include directive into the main files. This functionality is also supported for the VRML animation file processing.

Add Lights

In most animation sequences the lights will remain static. The lights are specified as *SpotLight* nodes in the input VRML file together with the description of the geometry of the actual light objects. The lights specifications are stored in the main file, while the geometry is contained in a separate geometry file. In order to add lights into a scene, it is unnecessary to undergo the computationally demanding process of generating the geometry data. Instead, only the light sources specifications can be inserted into the main file and the light geometry can be included as a static file. This functionality is also supported for the VRML animation file processing.

Duplicate Static Main File

As it was previously argued, most the layers of the MPII model will remain static during the animation sequence. However, every frame requires its own main file. This functionality takes a template main file and duplicates it for each frame of the animation. The content of the template file is copied and the file name is appropriately indexed. This functionality is also supported for the VRML animation file processing.

Add Animated Camera

Very often, the animations will contain certain modifications of the camera. In principle it could be only the camera that needs to be modified from a previous trial of the animation. The camera position and orientation is specified in the main file and no geometry data needs to be generated. The selected camera with its new settings is only added into the main files. This functionality is also supported for the VRML animation file processing.

Add Output File Name

The PBRT rendering engine enables for file name specification of the rendered images. This specification is stored in the header main file. In it essential for the animation post-processing to maintain appropriate file naming conventions and correct file indexing. Hence, this functionality allows for a simple modification of the output file name in each of the main files of the animation sequence. The file names are appropriately indexed.

Update Renderer

The PBRT main file contains the specification of the rendering algorithm and its properties. For instance, the resolution of the output image or the number of samples per pixel are specified here. Such parameters have a crucial influence on the quality of the output image but also on the computational complexity of the rendering process. It is very likely that users will need to experiment with different settings for the PBRT rendering engine. This can be easily performed by updating the rendering parameters stored in each main file of the animation sequence.

Linux Script Generation

Typical video sequences maintain around 25 frames per second. Despite considering only few seconds of video, a great number of individual frames must be rendered. In order to make this process as user friendly as possible, an automatic bash Linux script for scene rendering can be automatically generated. The script should control the number of scenes that are currently being rendered. Furthermore, in case of a system failure, the user should be able to easily modify the script to restart the rendering process from an arbitrary frame number.

At the beginning of the script, the index of the first scene to be rendered is specified. Next, a list of all scene files to be rendered by the PBRT application is given. The script iterates through all of the scenes and periodically checks if the desired number of scenes is currently being rendered. This is performed by listing the running processes of the user using the *ps* command and looking up all PBRT processes. If enough scenes are being rendered, then the script waits for a specified number of seconds. In case that an insufficient number of scenes was detected, the next scene from the list is passed to the PBRT rendering engine.

An example of the script with 6 scenes in the list, rendering 3 scenes at a time, waiting 60 second between two consecutive checks and starting from the third scene is shown below:

```
#!/bin/bash -x

limit=2
list="Anim_Final_1_F000.pbrt Anim_Final_1_F001.pbrt Anim_Final_1_F002.pbrt
Anim_Final_1_F003.pbrt Anim_Final_1_F004.pbrt Anim_Final_1_F005.pbrt "

i=0
for file in $list; do
    i=`expr $i + 1`
```



```

if [ $i -lt $limit ]; then
    continue;
fi
running=`ps | sed -n /pbprt/p |wc -l`
while [ $running -ge 2 ]; do
    sleep 20
    running=`ps | sed -n /pbprt/p | wc -l`
done
nohup pbprt $file > $file.log &
sleep 10
done

```

6.4.5 Animation Post-Processing

The implemented application generates input data for the PBRT rendering engine. The PBRT renders a sequence of individual frames. In order to obtain a video sequence an animation post-processing is required. This post-processing has two main stages. First, a batch tone mapping needs to be applied to the High Definition Range (HDR) images rendered by PBRT. Second, individual frames have to be converted into a single video sequence. Since the animation post-processing is not within the scope of this thesis, only the used software tools are briefly described below.

Batch Tone Mapping

The PBRT rendering engine takes provided input data and generates a sequence of images in HDR .exr format. While some software tools are capable of directly working with this format (e.g. Adobe Photoshop), it is not a suitable image format for creating a video sequence. Instead, the images need to be converted to a Low Dynamic Range (LDR) format first.

Despite many software tools for tone mapping being available, a one that also supports batch tone mapping is necessary for animation post-processing. The rendered animation sequences are commonly composed of a large amount of individual frames. Hence, the manual tone mapping of every single frame would be an inadmissible task. Software tools capable of batch tone mapping take a correctly indexed sequence of HDR images and automatically tone map all of them based on the selected settings.

Results presented in this thesis were obtained using the Qtpfsgui – Hdr Imaging Workflow Application [22], which supports batch tone mapping using one of many implemented tone mapping algorithms.

Image Sequence Conversion

The final step in the animation generation process is converting the indexed sequence of individual images (frames) into a video sequence. Several software tools are available for this task. The results presented in this thesis were produced using the VirtualDub application [23].

6.5 GUI Extensions

Several extensions to the GUI of the previous application were introduced in order to implement the new functionalities described in this thesis. This section illustrates and describes some of the major modifications: the display of available objects of interest, the dialog for setting the global animation parameters, the dialog for modifying and animating an object of interest, and the dialog for specifying the animation output.

Cameras Lights Doors Windows Shutters								
Door name	Type	Floor	Section	Rotation	Handle	Animated	Interpolation	
DOOR_01	3	6	2	0	0	False	Linear	
DOOR_02	4	6	2	0	0	False	Linear	
DOOR_06	3	6	1	0	0	False	Linear	
DOOR_07	3	6	1	0	0	False	Linear	
DOOR_09	3	6	1	0	0	False	Linear	
DOOR_10	3	6	1	0	0	False	Linear	
DOOR_12	3	6	1	0	0	False	Linear	
DOOR_14	3	6	1	0	0	False	Linear	
DOOR_08	4	6	1	0	0	False	Linear	
DOOR_11	4	6	1	0	0	False	Linear	
DOOR_13	4	6	1	0	0	False	Linear	
DOOR_15	4	6	1	0	0	False	Linear	
DOOR_19	4	6	4	0	0	False	Linear	
DOOR_20	4	6	4	0	0	False	Linear	
DOOR_18	3	6	4	0	0	False	Linear	
DOOR_21	4	6	3	0	0	False	Linear	
DOOR_22	4	6	3	0	0	False	Linear	
DOOR_24	4	6	3	0	0	False	Linear	
DOOR_26	4	6	3	0	0	False	Linear	
DOOR_23	3	6	3	0	0	False	Linear	
DOOR_25	3	6	3	0	0	False	Linear	
DOOR_04	3	6	2	0	0	False	Linear	
DOOR_05	3	6	2	0	0	False	Linear	
DOOR_03	4	6	2	0	0	False	Linear	
DOOR_36	1	6	2	0	0	False	Linear	
DOOR_37	1	6	2	0	0	False	Linear	
DOOR_38	1	6	2	0	0	False	Linear	
DOOR_39	1	6	2	0	0	False	Linear	
DOOR_27	3	6	2	0	0	False	Linear	
DOOR_35	1	6	2	0	0	False	Linear	
DOOR_33	1	6	2	0	0	False	Linear	
DOOR_31	1	6	2	0	0	False	Linear	

Fig. 45 Display of objects of interest.

Display of Objects of Interest

The main panel of the implemented application was modified to display the available objects of interest in the currently loaded dataset. Previously this panel only showed the available light sources. As can be seen in Fig. 45, the GUI now contains multiple panels showing the loaded camera, light, door, window and window shutter objects. Particular parameter values are displayed for each object. By clicking on a certain object, a modification dialog is invoked. This dialog is described below.

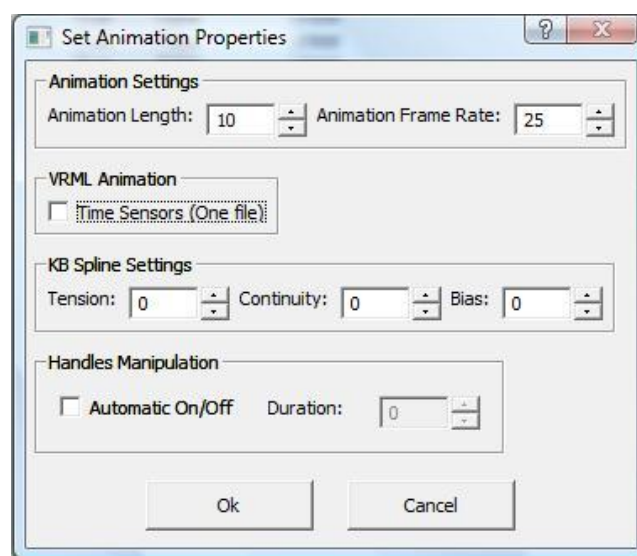


Fig. 46 Dialog for specifying the global animation properties.

Global Animation Settings Dialog

A new dialog was implemented for specification of the global animation properties. This dialog can be invoked from the main panel of the GUI and it is displayed in Fig. 46. The most important parameters that can be specified here are the length and the frame rate of the animation sequence.

Dialog for Modifying Objects of Interest

By clicking on a selected object of interest, a modification dialog is invoked. As an example, consider the dialog for modifying the camera object as shown in Fig. 47. The dialog displays the information about the camera object and enables modification of its static parameters as it was done before for the light objects. An animation sequence settings are now added into the GUI. This can be seen in the lower part of the figure. New animation nodes can be specified in the menu located on the right side and then added into the animation list. The animation list is displayed on the left. Animation nodes are listed in a sequence ordered according to the time along with their parameters. Also, the GUI allows for specification of the interpolation method to be used.

Dialog for Animation Output

A new dialogs were added for specifying the animation output for both PBRT and for VRML format. The dialog for PBRT animation output is depicted in Fig. 48. Part of this dialog resembles the previously implemented dialog for static scene PBRT output. However, a new section was added in the lower part of the dialog for modifying the main PBRT file. Here, the user can use any of the implemented file processing functionalities as described in Section 6.4.4.

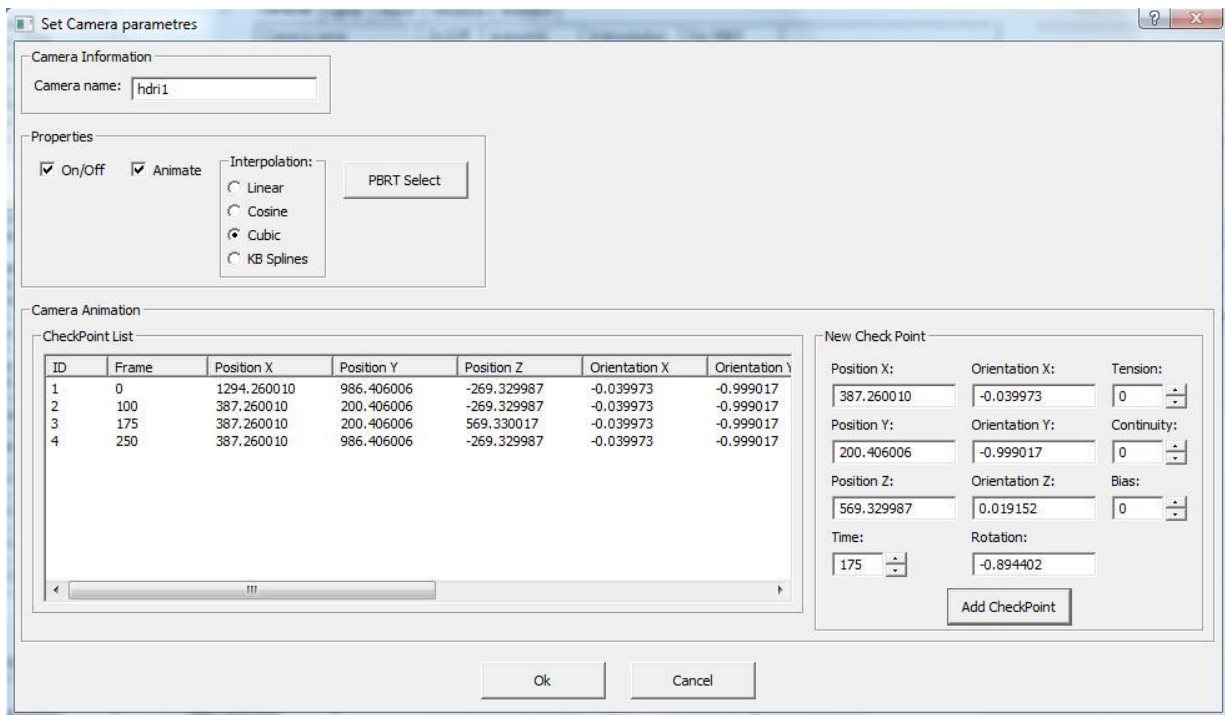


Fig. 47 Dialog for animating cameras.

6.6 Additional Extensions

The additional extensions – project status and console version of the VRMLtoPBRT application - have been implemented according to the proposed design from Section 5.4. As these extensions are not the primary scope of this thesis, their implementation is not described here in more detail. Appendix C presents an extensive description of the syntax of the project status textual format.

Create PBRT Animation

Select output directory Output directory: .

☐ Use absolute path for included files

☒ Generate Linux script

What kind of PBRT animation files you want create?

New PBRT files

Name of new PBRT animation file: pbrtAnim.pbrt

Insert render options?

☒ Yes ☐ No

To create individual PBRT animation files with light definitions and geometry in one file, click here: Create Animation

Separated PBRT animation files

Name of new PBRT animation main files: pbrtAnimMain.pbrt

Name of new PBRT animation geometry files: pbrtAnimGeom.pbrt

To create separated PBRT animation files with light definitions and render setting in one file and geometry in another file, click here: New Main New Geometry

Modify Main PBRT file

Base name of the PBRT main file: pbrtAnimMain.pbrt

Base name of the include / light source file: pbrtAnimGeom.pbrt

To modify the main file, click here:

Add Static Geometry Add Lights

Add Animated Geometry Duplicate Static Main File

Add Animated Camera Add Output File Name

Update Renderer

Back

Fig. 48 Dialog for specifying the PBRT animation output.

7 Results

This chapter presents the results of this thesis. The implemented geometry modifications of the MPII building model are demonstrated. Further, examples of the produced animations sequences are shown.

Due to the high computational complexity of the rendering task a school computer located on the Charles Square complex in the Virtual Reality lab was used. Its hardware specifications are as follows:

- Two Quad processors Intel(R) Xeon(R) CPU E5440 at 2.83 GHz
- 48 GB RAM

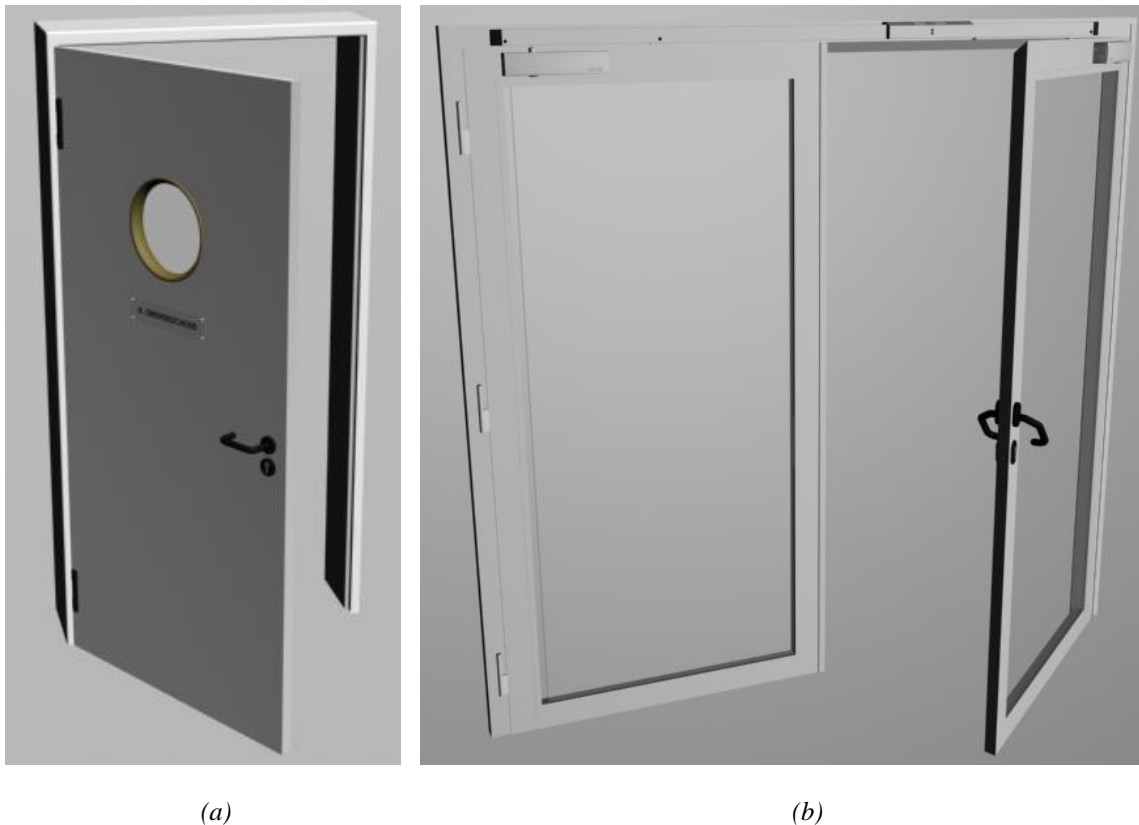


Fig. 49 Geometry modification of two types of door objects.

7.1 Geometry Modifications

The geometry modifications have been implemented as described earlier in Chapters 5 and 6. The purpose of this section is to visually demonstrate the correctness and accuracy of the geometry modifications applied to the MPII data model. Later on the correctness of geometry modifications can be verified on the produced animation sequences.

The snapshot of animation for two door objects can be seen in Fig. 49. The opening of a single door is shown in Fig. 49(a) followed by the opening of a double glass door in Fig. 49(b). Next, the opening of the window object is displayed in Fig. 50. Both the vertical and the horizontal opening of windows with appropriately rotate window handles can be observed. Finally, the two rates of shutter opening can be seen in Fig. 51.



Fig. 50 Geometry modification of the window object.

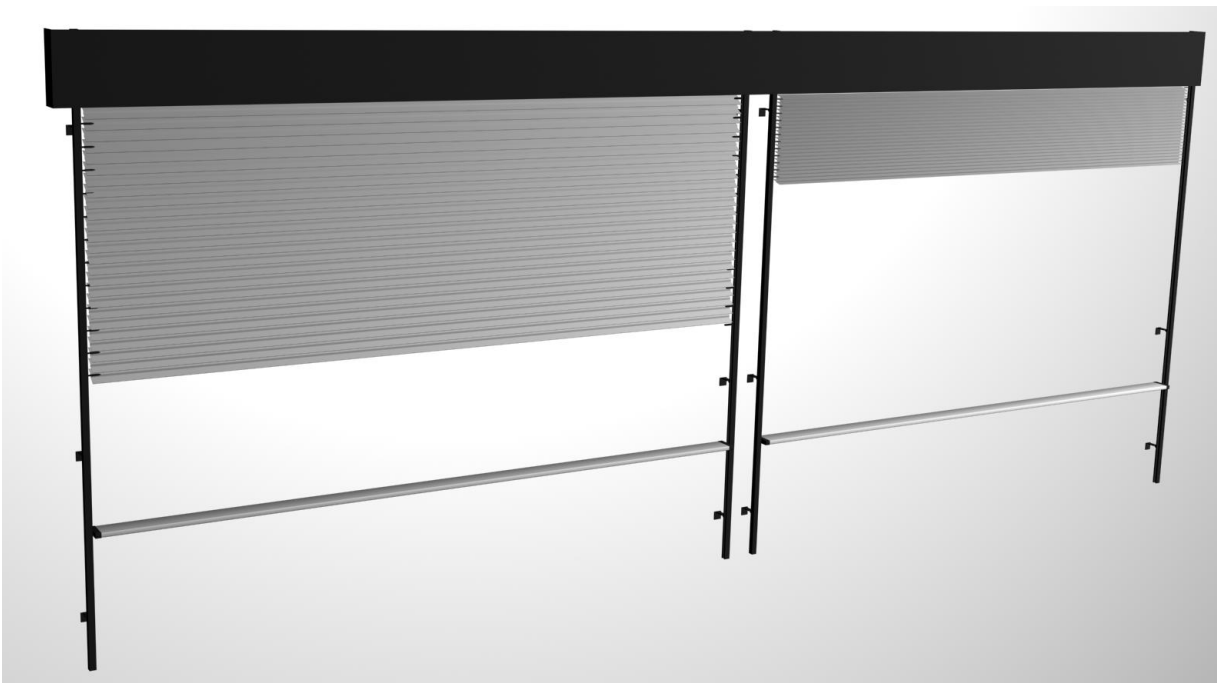


Fig. 51 Geometry modification of the window shutter object.

7.2 Animations

As it was specified in the abstract of this thesis, five animation sequences of length 10 seconds were to be created in order to demonstrate the correctness of the application design and implementation. This section describes the created animations.

The designed animations tested both static and dynamic geometry modifications, various camera animation techniques in different parts of the MPPII building. Each animation sequence was 10 second long with frame rate of 25 fps. Hence, the animation preprocessing required preparation of 250 scene descriptions for the PBRT rendering engine. Due to the immense computational time required for rendering of the generated scenes, the complexity of the rendering algorithm was set slightly lower than it was used in previous work for generating the final images. Below is a description of the PBRT render settings used for the first four animations:

- Image resolution: 1024 x 768
- Surface integrator: Whitted ray-tracer, maximum recursion depth: 5
- Sampler: Best candidate, pixel samples: 5
- Pixel Filter: Mitchell, B: 0.33, C: 0.33, x-width: 2.0, y-width: 2.0

The processing quality of the fifth animations had to be reduced because of time constraints. It was rendered with the following settings:

- Image resolution: 1024 x 768
- Surface integrator: Whitted ray-tracer, maximum recursion depth: 3
- Sampler: Best candidate, pixel samples: 2
- Pixel Filter: Mitchell, B: 0.33, C: 0.33, x-width: 2.0, y-width: 2.0

Each of the five produced animation sequences is described below:

7.2.1 Animation 1 – Static Camera in the Hall

This animation sequence verified the correctness of the implemented geometry modifications. The animation starts at the beginning of the hall on the 6th floor. The camera remains static the whole time. During the animation, the doors in the hall open and close in a sequence, starting close to the camera and finishing with the last doors in the hall. Opening of a door is preceded by turning the door handle by a 45 degrees angle.

By rendering three frames in parallel, the rendering time of this animation was 2 days, 14 hours, 20 minutes and 17 seconds. Snapshots from this animation are shown in Fig. 52.

7.2.2 Animation 2 – Dynamic Camera in the Hall

This animation was based on the geometry of Animation 1, but it added dynamic camera movement. The animation also starts at the beginning of the hall on the 6th floor. During the animation the doors in the hall are opened in sequence in the same manner as in Animation 1. The camera starts moving through the hall and it turns the view to the right towards the glass door into the atrium at the end of the hall.

By rendering three frames in parallel, the rendering time of this animation was 3 days, 42 minutes and 53 seconds. Snapshots from this animation are shown in Fig. 53.

7.2.3 Animation 3 – Into the Atrium

This animation sequence starts at the position and camera orientation where the previous animation finished. The animation begins by opening the glass door in front of the camera. First, the door handle is rotated by 45 degrees. After the door is opened, the camera moves through the door into the atrium. During the movement, the camera slowly rotates to the right and shows the whole atrium.

By rendering three frames in parallel, the rendering time of this animation was 8 days, 12 hours, 14 minutes and 24 seconds. Snapshots from this animation are shown in Fig. 54.

7.2.4 Animation 4 – Entrance

This animation sequence starts with the view of the main rotational door. The rotational door is spinning. The camera slowly rotates pass the rotational doors and shows the ground floor of the atrium of the MPII building.

By rendering three frames in parallel, the rendering time of this animation was 22 days, 12 hours, 37 minutes and 37 seconds. Snapshots from this animation are shown in Fig. 55.

7.2.5 Animation 5 – Roundel

This final animation shows a view of the MPII atrium as seen from the roundel room on the 2nd floor. The camera first slowly moves towards the windows of the roundel and then smoothly rotates to the right to show the whole atrium.

By rendering three frames in parallel, the rendering time of this animation was 10 days, 12 hours, 37 minutes and 37 seconds. Snapshots from this animation are shown in Fig. 56.



(a)



(b)



(c)



(d)



(e)



(f)

Fig. 52 Snapshots from the Animation 1- Static camera in the hall.



(a)



(b)



(c)



(d)



(e)



(f)

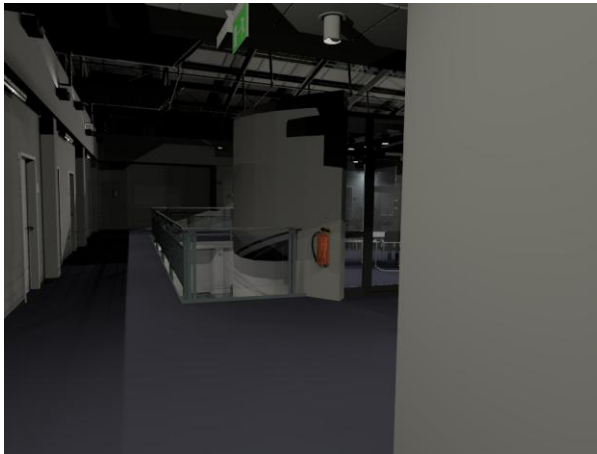
Fig. 53 Snapshots from Animation 2 – Dynamic camera in the hall.



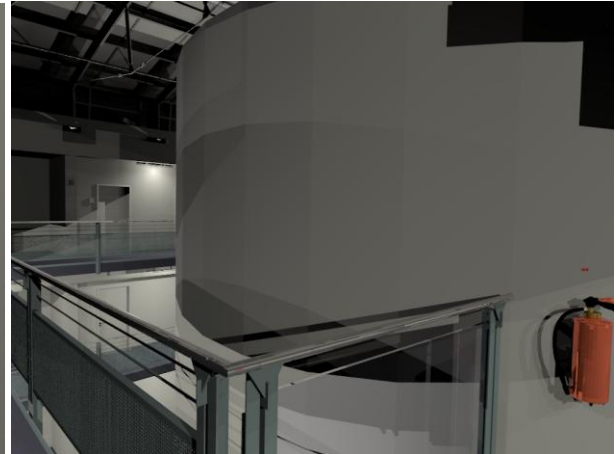
(a)



(b)



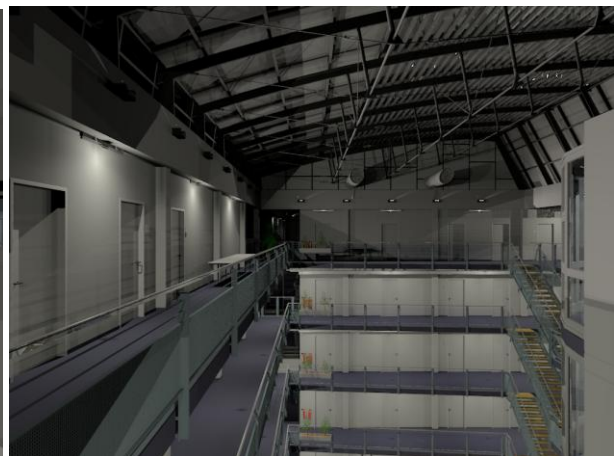
(c)



(d)

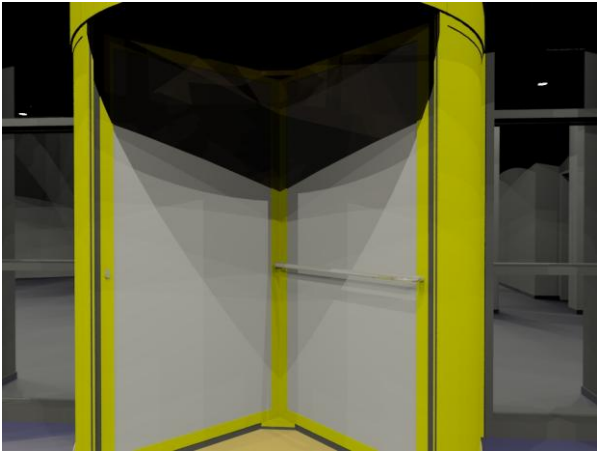


(e)

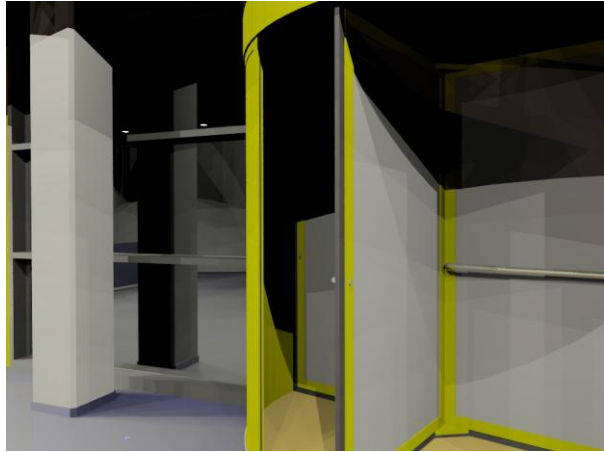


(f)

Fig. 54 Snapshots from Animation 3 – Into the Atrium.



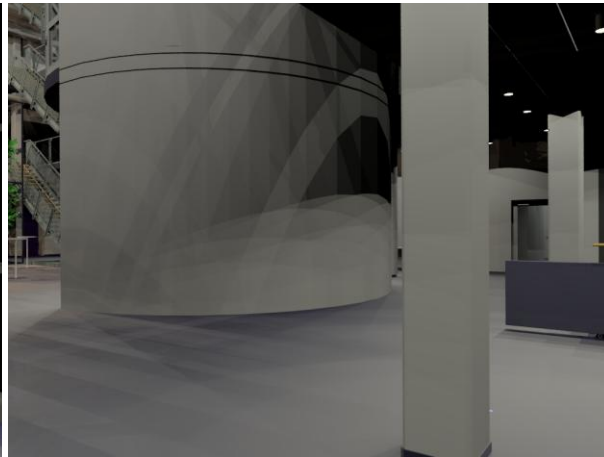
(a)



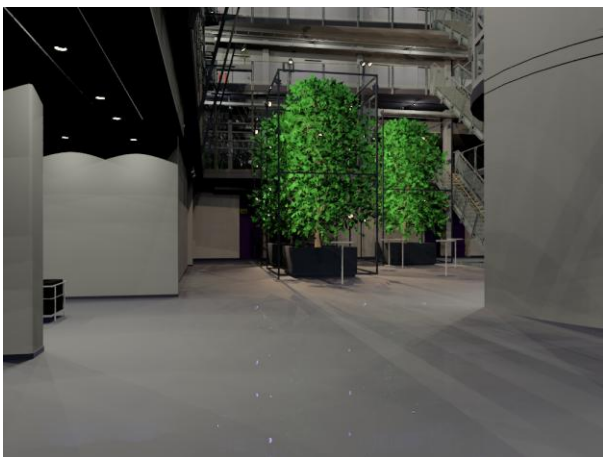
(b)



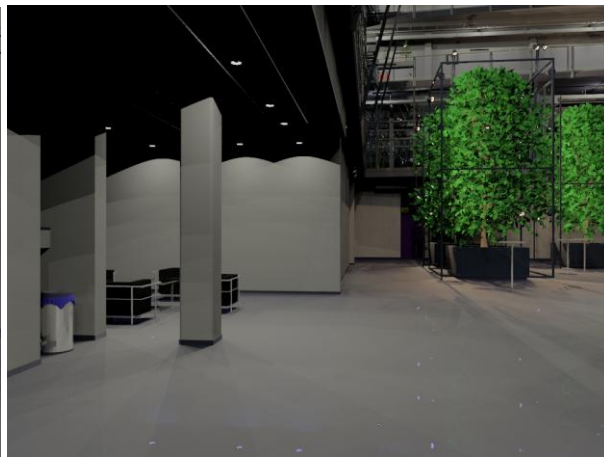
(c)



(d)



(e)

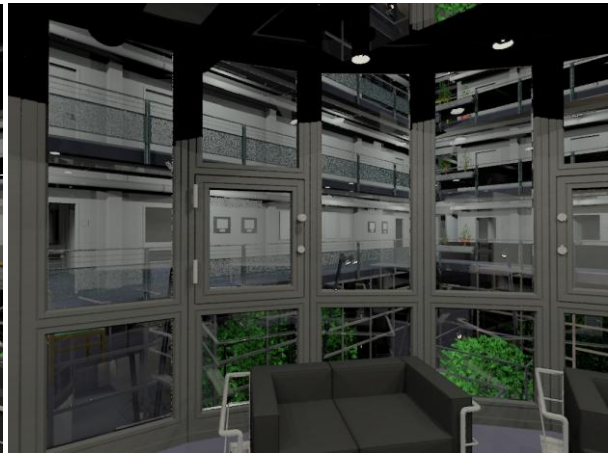


(f)

Fig. 55 Snapshots from Animation 4 – Entrance.



(a)



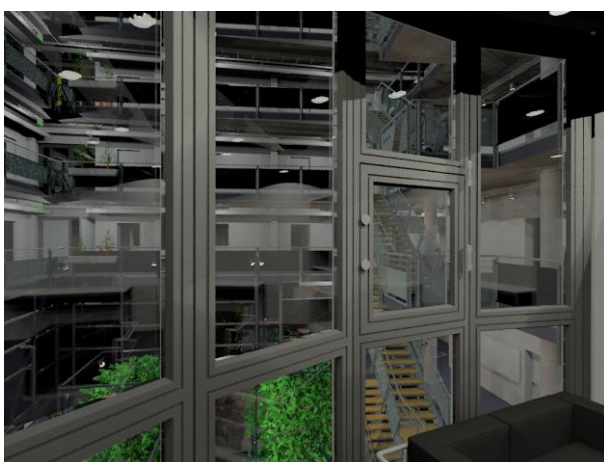
(b)



(c)



(d)



(e)



(f)

Fig. 56 Snapshots from Animation 5 – Roundel.

8 Conclusion

This thesis extended the previous work on the MPII data model for predictive image synthesis. The original geometry of the MPII model was parameterized to allow for modifications of the movable parts of the model such as doors, windows, window shutters and their auxiliary parts. In addition, the previously implemented VRMLtoPBRT application was extended to contain control elements supporting the geometry modifications. Finally, animation processing was implemented in the VRMLtoPBRT application. The implemented work can be divided into two categories: i) modification of the MPII data model, and ii) extending the VRMLtoPBRT application.

The modification of the MPII data model can be summarized as follows:

- Designing object hierarchies for each object of interest in 3DS Max.
- Rebuilding the object hierarchies of all objects of interest in the MPII data model according to the design.
- Appropriate labeling of all objects of interest and their individual parts.
- Addition of lights sources into the offices on the 2nd – 6th floors.
- Removal of geometry errors from the 3DS Max model (geometry errors due to incorrect VRML export of the mirror transformations).

The major implemented extensions to the VRMLtoPBRT application are summarized below:

- Decoding of the appropriately labeled objects of interest and their displaying in the GUI of the application.
- GUI dialogs for specification of the geometry modifications of certain object of interest.
- Implementation of the global animation object, which synchronizes the animation processing of all objects of interest.
- GUI dialogs for animating the parameters of selected objects of interest.
- Implementation of supporting functionalities for animation processing. The implemented functions substantially reduce the redundancy in the output data generation process.
- Implementation of savable projects status, which logs the sequence of geometry modifications performed by the user, including the input data and the output actions.
- Implementation of project list processing, which allows for automatic processing of multiple projects specified in a single text file.
- Implementation of a console version of the VRMLtoPBRT application, independent of the wxWidgets library.

The correctness of the proposed approaches and the validity of the implementation were verified by producing 5 animation sequences. The animation sequences show geometry modifications of the MPII data model as well as camera animations in different parts of the MPII building.

Future extensions to the project can comprise any of the followings:


- Correction of additional geometry inconsistencies in the MPII data model. This can be done either via remodeling the MPII 3DS Max model or by implementing a correct VRML exporter for the 3DS Max application.
- Increasing the complexity and realism of the MPII data model.
- Acquisition of the true BRDFs of the materials existent in the MPII building.

9 References

- [1] J. Drahokoupil, “Prediktivní syntéza obrazu z modelu budovy MPII,” Master Thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, Czech Republic, 2009. (in Czech)
- [2] F. Drago, K. Myszkowski, “Validation proposal for global illumination and rendering techniques,” *Computer & Graphics*, vol 25, pp. 511-518, 2001.
- [3] Senate of the Max Planck Society, “Rules of Good Scientific Practice,” [URL], Available: <http://www.mpg.de/pdf/rulesScientificPract.pdf>, from November 2009.
- [4] Cornell box [URL], Available: <http://www.graphics.cornell.edu/online/box/>. Cornell University, 1998.
- [5] J. Žára, B. Beneš, J. Sochor, P. Felkel. *Moderní počítačová grafika*, Computer Press, Brno, 2th edition, 2004. ISBN 80-251-0454-0.
- [6] B. T. Phong, “Illumination for Computer Generated Images,” *Communication with the ACM*, 18(6), June 1975.
- [7] N. Metropolis, The beginning of the Monte Carlo method, Los Alamos Science Special Issue, 1987.
- [8] Columbia-Utrecht Reflectance and Texture Database [URL], Available: <http://www1.cs.columbia.edu/CAVE/software/curet/>, from November 2009.
- [9] Julian Smart, Kevin Hock, Stefan Csomar. *Cross-Platform GUI Programming with wxWidgets*, Prentice Hall PTR, 2005. ISBN 0-13-147-381-6.
- [10] Jiří Žára. VRML - Laskavý průvodce virtuálními světy. Computer Press, Brno, 1999. ISBN 80-7226-143-6.
- [11] 3D Modeling [URL], Available: http://en.wikipedia.org/wiki/3D_modeling, from November 2009.
- [12] 3D Studio Max Official website [URL], Available: <http://usa.autodesk.com>, from November 2009.
- [13] VRML 2.0 Standard Specification [URL], Available: <http://www.graphcomp.com/info/specs/sgi/vrml/spec/>, from November 2009
- [14] Kochanek-Bartels Spline [URL], Available: http://en.wikipedia.org/wiki/Kochanek%E2%80%93Bartels_spline, from December 2009.
- [15] D. Elberly, “Kochanek-Bartels Cubic Splines (TCB Splines),” *Geometric Tools Documentation*, February 2008.
- [16] M. Pharr, G. Humphreys, *Physically Based Rendering – From Theory to Implementation*, Morgan Kaufman Publishers, San Francisco, USA, 2004, ISBN 0-12-553180-X.

- [17] T. Whitted, "An Improved Illumination Model for Shaded Display," *ACM Communication on Graphics*, 1980.
- [18] J. T. Kajiya, "The Rendering Equation," *In SIGGRAPH'86 Conference Proceedings*, pp. 143-150, August 1986.
- [19] C. M. Gora, K. E. Torrance, D. P. Greenberg, B. Battaile, "Modelling the Interaction of Light Between Diffuse Surfaces," *in SIGGRAPH '84 Conference Proceedings*, pp. 212-222, July 1984.
- [20] A. Keller, "Instant radiosity," *in SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 49-56, New York, NY, USA, 1997.
- [21] Cortona 3D Main page [URL], Available: <http://www.cortona3d.com/>, from December 2009.
- [22] QtPfsGui Home page [URL], Available: <http://qtpfsgui.sourceforge.net>, from January 2010.
- [23] VirtualDub Home page [URL], Available: <http://www.virtualdub.org/index>, from January 2010.
- [24] V. Havran, J. Zajac, J. Drahokoupil, H.-P. Seidel, "MPI Building Model as Data for Your Research," *Technical report MPI-I-2009-4-004*, December 2009.

Appendix A – Light Types Corrections

Light Image	Light Name	EXR File Name	Model Id	Thesis Id
	Louis Poulsen	IESlouis_poulsen.exr	1	1
	Siemens 5LJ180	IESsiemens.exr	3	2
	Radium NL	IESradium.exr	5	3
	RJH-TS Halogen Spot HL500	IESrjh_ts.exr	0	4
	Philips	IESphilips.exr	4	5
	Osram CONCENTRA PAR 38	IESosram.exr	6	6

Appendix B – List of Materials

1. BF_louisPoulsen	
Usage	Material used for transparent lamp-shade of Louis Poulsen luminaires (luminaire # 1).
2. BF_Alum01	
Usage	Shiny alluminium surfaces at the edge of white boards, the metallic dustbins.
3. BF_black01	
Usage	Material used for plastic parts (washers, heelpieces), matte plastic.
4. BF_black02	
Usage	Material used for the chair body (backrest and seat) for shiny black chairs.
5. BF_black03	
Usage	Material for black leathered seats and chairs.
6. BF_black04	
Usage	Material used for metallic parts of furniture.
7. BF_black05	
Usage	Material use for black gum surface, for example for grab handle of fire extinguishers.
8. BF_blue01	
Usage	Material use for plastic heelpieces and jointings, for example white boards.
9. BF_blue02	
Usage	Material used for plastic chairs in the coffee bar in the first floor.
10. BF_bolts01	
Usage	Material used for metallic screws and bolts.
11. BF_brMetal01	
Usage	Material used for grab handles, gribs, and chair frames.
12. BF_brown01	
Usage	Material used as filling of cork boards.
13. BF_cream01	
Usage	Material used for the frames of green boards.
14. BF_foxiness01	
Usage	Material used for throttle-valves of fire extinguishers and heating parts (cock-metal).
15. BF_glass01	
Usage	Material used as glass filling of a round table.
16. BF_gray01	
Usage	Material used for metallic parts of stair-rails and other constructions.
17. BF_gray02	

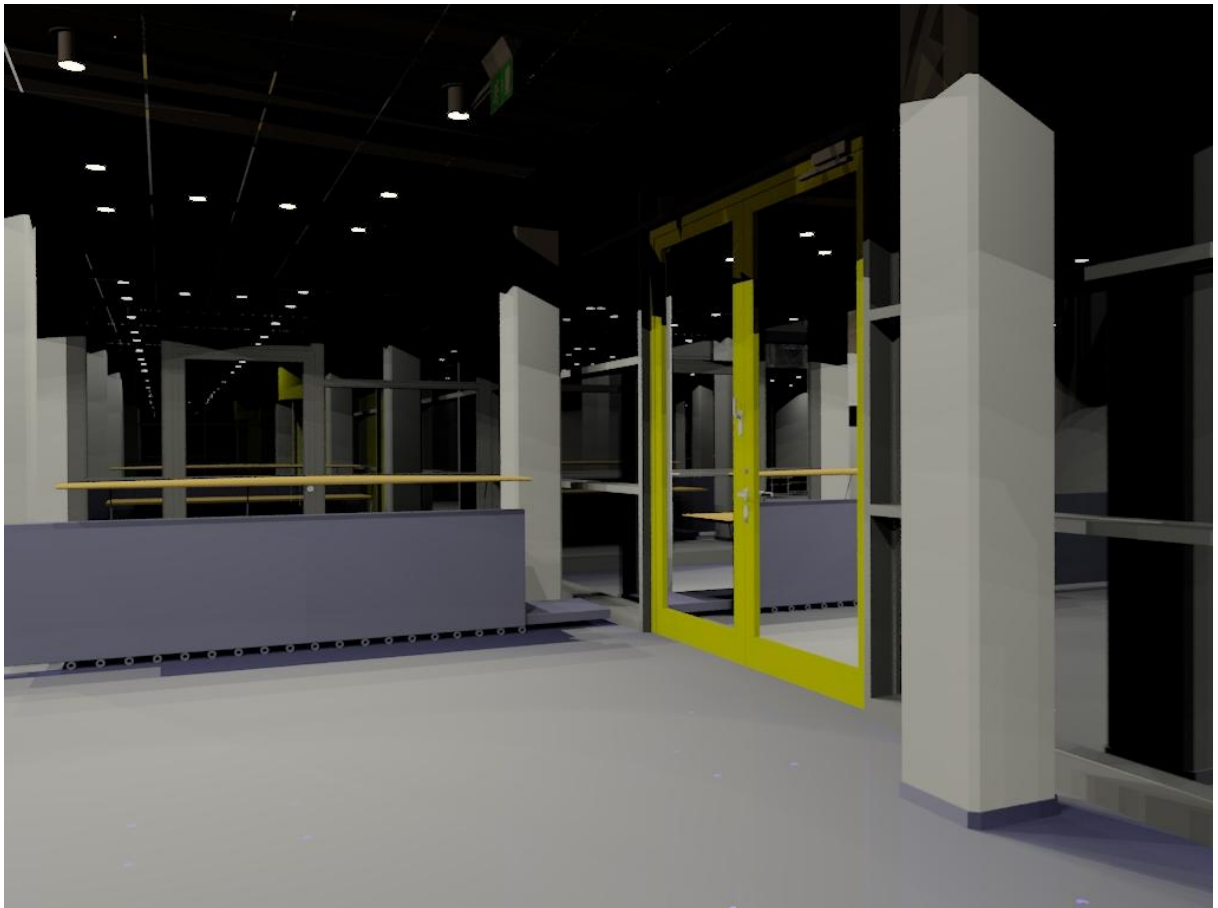
Usage	Material used for support of office furniture.
18. BF_gray03	
Usage	Material used for plastic backrest of metallic sitting benches.
19. BF_gray04	
Usage	Material used for majority of office furniture surfaces.
20. BF_green01	
Usage	Material used for the interior of green boards.
21. BF_green02	
Usage	Material used for the seat filling of green chairs in the rondell.
22. BF_green03	
Usage	Material used for plastic chairs in the coffee bar in the first floor.
23. BF_red01	
Usage	Material used for the metallic body of fire extinguishers.
24. BF_red02	
Usage	Material used for the metallic body of fire extinguishers (CO ₂).
25. BF_red03	
Usage	Material used for plastic chairs in the coffee bar in the first floor.
26. BF_redBrown01	
Usage	Material used for the bolstering of office wheel-chairs (backrest and seat).
27. BF_shMetal01	
Usage	Material used for metallic parts of chairs (chair-frame).
28. BF_white01	
Usage	Material used as the filling of white boards.
29. BF_white02	
Usage	Material used as the trestleboard of the solid round table.
30. BF_wood01	
Usage	Material used for the wooden parts of sitting chairs in the rondell.
31. BF_yellow01	
Usage	Material used for plastic chairs in the coffee bar in the first floor.
32. CH_alum01	
Usage	Material used for the frame holders of luminaires in the rondell.
33. CH_alum02	
Usage	Material used for metallic parts of ventilation.
34. CH_alum03	
Usage	Material used for metallic doors and parts of the lift.
35. CH_black01	
Usage	Material used for the seats of backrest of chesterfields and couches.
36. CH_black02	
Usage	Material used for the door pulling handle made of metal.
37. CH_constructionC01	

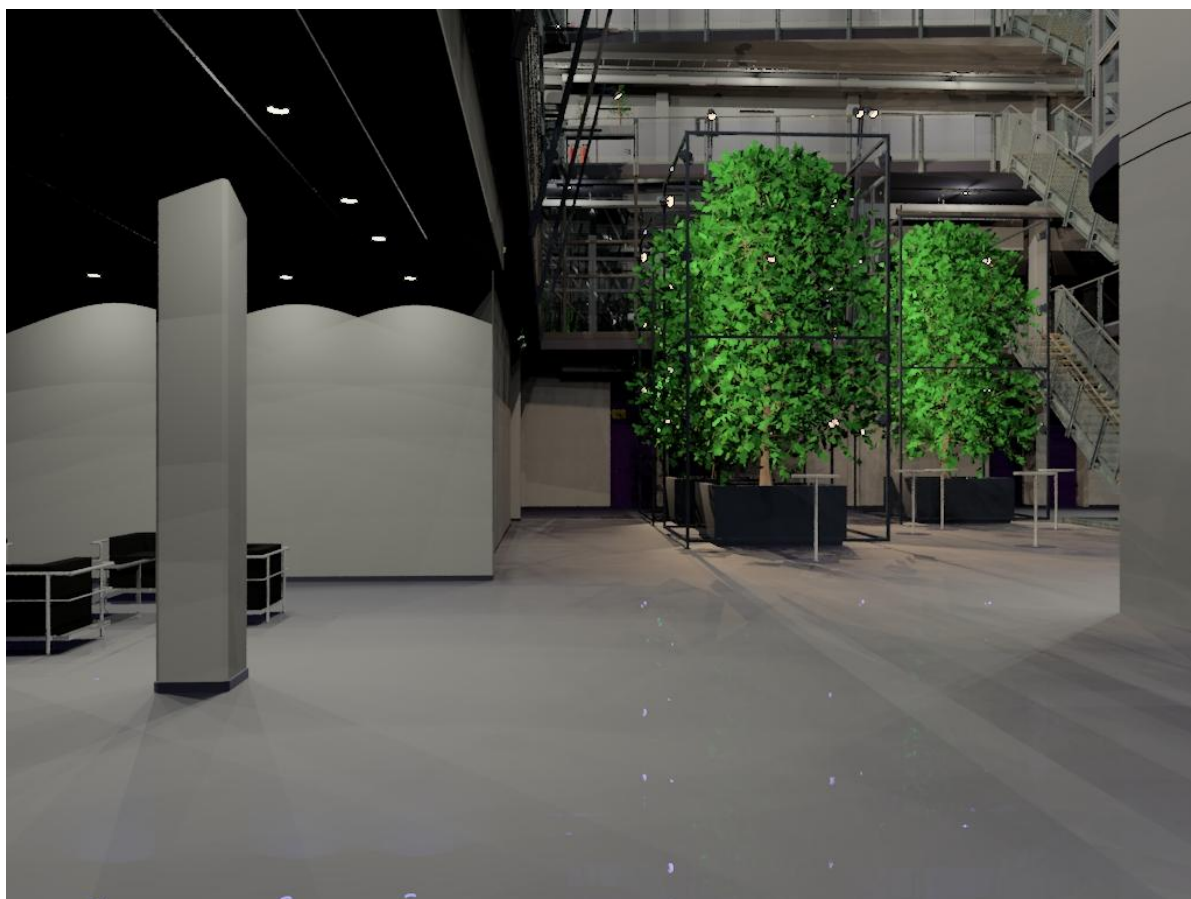
<i>Usage</i>	Material used for metallic frames around the plants and trees in the ground floor.
38. CH_glass01	
<i>Usage</i>	Material used as glass for the doors and windows.
39. CH_glass02	
<i>Usage</i>	Material used as glass for the filling in handrails.
40. CH_gray02	
<i>Usage</i>	Material used for the office doors.
41. CH_gray03	
<i>Usage</i>	Material used as the carpet in the offices and corridors.
42. CH_green01	
<i>Usage</i>	Material used for the leaves of trees and plants.
43. CH_metal01	
<i>Usage</i>	Material used for the metallic ledger-board of handrails, in particularly for the stairs and corridors.
44. CH_pavers01	
<i>Usage</i>	Material used for the tiles (flagstone) in the ground floor (perhaps granite).
45. CH_violet01	
<i>Usage</i>	Material used for some walls in the ground floor.
46. CH_white01	
<i>Usage</i>	Material used for the plaster of walls in the whole building interior.
47. CH_white02	
<i>Usage</i>	Material used for the metallic frames of doors, the plastic cables isolations, the fire alarms, and some parts of fluerescent tubes.
48. CH_white04	
<i>Usage</i>	Material used for the celing panels in the ground floor.
49. CH_white05	
<i>Usage</i>	Material used for the central heating bodies.
50. CH_yellow01	
<i>Usage</i>	Material used for the front revolving doors in the ground floor and for the other metallic construction outside.
51. CH_yellow02	
<i>Usage</i>	Material for the wooden stair-case treads (steps) between the for the main stair-case.
52. CH_yellow03	
<i>Usage</i>	Material used for the floor in the revolving doors in the ground floor.

Appendix C – Image Gallery













Appendix D – Project Status File Format

This appendix discusses the file format for storing the project status of the VRMLtoPBRT application. The implemented system uses a simple methodology, where each line in the project file represents a specific command. Each command is denoted by a specific prefix, followed by its name. After the command name, a list of parameters follows. The prefix of each command denotes the category of each command. The list of used prefixes is listed in Table 5.

Next, Table 6 summarizes all the commands used to describe the loaded dataset, parameters of the renderer and the global animation object as well as individual geometry modification actions of specific objects of interest. Each entry in the table shows the name of the command, its description and explanation of all parameters. An example of specific command is provided for each entry. The parameters are labeled by characters with respect to their data types (I – integer, F – float, S - string).

Finally, the implemented output functions are specified in Table 7. All of the output functions use the same command. The first two parameters of the output command differentiate between various output functions. Again, each entry in the table lists the name of the command, its description, explanation of its parameters and an example of its use.

Table 5 List of prefix categories of the project file format.

Prefix	Meaning
CH_	Input data description and checking information
R_	PBRT renderer settings
GA_	Global animation settings
A_	Geometry modifications and animation actions
O_	Output actions

Table 6 Description of the project file format commands.

CH_DataFile	
<i>Description</i>	File name of the input VRML file to be loaded
<i>Parameters</i>	S1 – file path
<i>Example</i>	CH_DataFile C:\Ondrej\TestScenes\0-6_lamps_3.WRL
CH_Dataset	
<i>Description</i>	Checking parameters describing the input VRML dataset
<i>Parameters</i>	I1 – number of nodes, I2 - number of base nodes, I3 – number of cameras, I4 – number of lights, I5 – number of doors, I6 – number of windows, I7 – number of window shutters
<i>Example</i>	CH_Dataset 78 11 25 0 0 0 0
R_Resolution	
<i>Description</i>	Resolution of the PBRT rendered image
<i>Parameters</i>	I1 – X resolution, I2 – Y resolution
<i>Example</i>	R_Resolution 500 500
R_FileName	
<i>Description</i>	File name of the output PBRT image
<i>Parameters</i>	S1 – file name

<i>Example</i>	R_FileName my_render.exr
R_FilePath	
<i>Description</i>	File path for the output PBRT images
<i>Parameters</i>	S1 – file path
<i>Example</i>	R_FilePath ./Renders/
R_RT	
<i>Description</i>	Maximum recursion depth for the Ray-tracing algorithm
<i>Parameters</i>	I1 – maximum recursion depth
<i>Example</i>	R_RT 5
R_PT	
<i>Description</i>	Maximum recursion depth for the Path-tracing algorithm
<i>Parameters</i>	I1 – maximum recursion depth
<i>Example</i>	R_PT 5
R_IGI	
<i>Description</i>	Parameter specification for the Instant Global Illumination algorithm
<i>Parameters</i>	I1 – maximum recursion depth, I2 – number of paths from lights, I3 – number of light sets, F4 – maximum distance to a light, F5 – Russian roulette threshold, F6 – indirect scale
<i>Example</i>	R_IGI 5 10 4 0.1 1.0 1.0
R_BestCandidate	
<i>Description</i>	Description of the best candidate sampler
<i>Parameters</i>	I1 – number of pixel samples
<i>Example</i>	R_BestCandidate 5
R_LowDiscrepancy	
<i>Description</i>	Description of the low discrepancy sampler
<i>Parameters</i>	I1 – number of pixel samples
<i>Example</i>	R_LowDiscrepancy 5
R_Stratified	
<i>Description</i>	Description of the stratified sampler
<i>Parameters</i>	I1 – number of samples along X axis, I2 – number of samples along Y axis, I3 – Jitter on/off
<i>Example</i>	R_Stratified 2 2 1
R_Box	
<i>Description</i>	Description of the box pixel filter
<i>Parameters</i>	F1 – X width, F2 – Y width
<i>Example</i>	R_Box 2.0 2.0
R_Gaussian	
<i>Description</i>	Description of the Gaussian pixel filter
<i>Parameters</i>	F1 – X width, F2 – Y width, F3 – alpha parameter
<i>Example</i>	R_Gaussian 2.0 2.0 2.0
R_Mitchel	
<i>Description</i>	Description of the Mitchel pixel filter
<i>Parameters</i>	F1 – X width, F2 – Y width, F3 – B parameter, F4 – C – parameter
<i>Example</i>	R_Mitchel 2.0 2.0 0.3 0.3

R_Triangle	
<i>Description</i>	Description of the triangle pixel filter
<i>Parameters</i>	F1 – X width, F2- Y width
<i>Example</i>	R_Triangle 2.0 2.0
GA_AnimLength	
<i>Description</i>	Length of the animation sequence
<i>Parameters</i>	F1 – length of the animation sequence in seconds
<i>Example</i>	GA_AnimLength 5.0
GA_AnimFPS	
<i>Description</i>	Frame rate of the animation sequence in frames per second
<i>Parameters</i>	F1 – frame rate per second
<i>Example</i>	GA_AnimFPS 10.0
GA_AnimKB	
<i>Description</i>	Description of the global parameters for the KB-splice interpolation
<i>Parameters</i>	F1 – tension, F2 – continuity, F3 – bias
<i>Example</i>	GA_AnimKB 0.0 0.0 0.0
GA_AnimHandles	
<i>Description</i>	Description of the automatic handles manipulation
<i>Parameters</i>	I1 – automatic handles manipulation on/off, I2 – duration of automatic handles manipulation in frames
<i>Example</i>	GA_AnimHandles 1 15
GA_VRMLExpand	
<i>Description</i>	Description of the VRML format to be used
<i>Parameters</i>	I1 – Extended VRML format on/off
<i>Example</i>	GA_VRMLExpand 1
GA_MatFile	
<i>Description</i>	Specified the material file to be loaded
<i>Parameters</i>	S1 – Path and file name of the material file
<i>Example</i>	GA_MatFile ./materials/Materials.mat
A_Door	
<i>Description</i>	Description of the door object geometry modification action
<i>Parameters</i>	I1 – application mode (single, section, floor, all), I2 – door index, I3 – opening rate, I4 – handle opening rate, I5 – animation flag – the rest of the parameters is optional based on the animation flag, I6 – index of the interpolation method used, I7 – number of inserted animation nodes, for each animation node{I8 – animation time, I9 – door open rate, I10 – door handle open rate}
<i>Example</i>	A_Door 0 45 75 45 0 A_Door 0 45 0 0 1 1 2 0 0 0 10 75 45
A_Window	
<i>Description</i>	Description of the window object geometry modification action
<i>Parameters</i>	I1 – application mode (single, section, floor, all), I2 – window index, I3 – opening rate, I4 – handle opening rate, I5 – opening mode, I6 – animation flag – the rest of the parameters is optional based on the animation flag, I7 – index of the interpolation

	method used, I8 – number of inserted animation nodes, for each animation node {I9 – animation time, I10 – window open rate, I11 – window handle open rate, I12 – opening mode}
<i>Example</i>	A_Window 0 31 75 45 0 0 A_Window 0 31 0 0 0 1 1 2 0 0 0 1 10 55 100 1
A_Shutter	
<i>Description</i>	Description of the window shutter object geometry modification action
<i>Parameters</i>	I1 – application mode (single, section, floor, all), I2 – shutter index, I3 – scaling rate, I4 – rotation rate, I5 – animation flag – the rest of the parameters is optional based on the animation flag, I6 – index of the interpolation method used, I7 – number of inserted animation nodes, for each animation node {I8 – animation time, I9 – shutter scaling rate, I10 – shutter rotation rate }
<i>Example</i>	A_Shutter 0 32 50 100 0 A_Shutter 0 32 0 0 1 1 2 0 0 0 10 50 100
A_LightMod	
<i>Description</i>	Description of light animation
<i>Parameters</i>	I1 – application mode (single, section, floor, all), I2 – light object index, S3 – light name, I4 – light type, I5 – floor location, I6- section, I7 – on/off flag, I8 – IES assigned yes/no, S9 – IES file name, F10 – red component of the light color, F11- green component of the light color, F12 – blue component of the light color, F13 – light intensity, F14 – light depletion factor, I15- animation flag - the rest of the parameters is optional based on the animation flag, I16 – index of the interpolation method used, I17 – number of inserted animation nodes, for each animation node {I18 – animation time, I19 – light on/off flag, I20 light intensity}
<i>Example</i>	A_LightMod 0 0 Light_01 1 0 2 1 1 IESosram.exr 1.0 1.0 1.0 1.0 1.0
A_Camera	
<i>Description</i>	Description of the camera animation
<i>Parameters</i>	I1 - application mode (single, section, floor, all), I2 – camera index, I3 – camera on/off, I4 – camera selected for PBRT rendering yes/no, I5 - animation flag - the rest of the parameters is optional based on the animation flag, I6 – index of the interpolation method used, I7 – number of inserted animation nodes, for each animation node { I8 – animation time, F9 – X camera position, F10 – Y camera position, F11 – Z camera position, F12 – X camera orientation, F13 – Y camera orientation, F14 – Z camera orientation, F15 – angle of camera orientation around the specified axis, F16 – tension parameter of the KB interpolation, F17 – continuity parameter of the KB interpolation, F18 – bias parameter of the KB interpolation}
<i>Example</i>	A_Camera 0 14 1 1 0
A_LightSwitch	
<i>Description</i>	Turning of multiple lights on or off
<i>Parameter</i>	I1 - application mode (single, section, floor, all), I2 – light on/off,

	I3 – floor index, I4 – section number
<i>Example</i>	A_LightSwitch 1 1 5 2
A_LightType	
<i>Description</i>	Modifying the static properties of lights with certain type
<i>Parameter</i>	I1 - application mode (setting the EXR, color, or radiance), I2 – light type, S3 – file name of the EXR texture, F4 – red component of the light color, F5 – green component of the light color, F6 – blue component of the light color, F7 – radiance value
<i>Example</i>	A_LightType 1 1 unknown 1.0 0.0 0.0 0.0

Table 7 List of output actions of the project file format.

O_OutputAction 0	
<i>Description</i>	Exporting single static scene into VRML
<i>Parameters</i>	S1 – file path and name of the output VRML file
<i>Example</i>	O_OutputAction 0 ./scene1.wrl
O_OutputAction 1 0	
<i>Description</i>	Exporting single static scene into PBRT – header and geometry in one individual file
<i>Parameters</i>	S1 – file path and name of the output PBRT file
<i>Example</i>	O_OutputAction 1 0 ./scene1.pbrt
O_OutputAction 1 1	
<i>Description</i>	Exporting single static scene into PBRT – header and geometry in separate files, the header file is being created
<i>Parameters</i>	S1 – file path and name of the PBRT header file, S2 – file path and name of the PBRT geometry file, I3– use absolute path yes/no
<i>Example</i>	O_OutputAction 1 1 ./sceneHeader.pbrt ./sceneGeom.pbrt 0
O_OutputAction 1 2	
<i>Description</i>	Exporting single static scene into PBRT - header and geometry in separate files, the geometry is appended into a previously existing header file
<i>Parameters</i>	S1 – file path and name of the PBRT header file, S2 – file path and name of the PBRT geometry file, I3 – use absolute path yes/no
<i>Example</i>	O_OutputAction 1 2 ./sceneHeader.pbrt ./sceneGeom.pbrt 0
O_OutputAction 2 0	
<i>Description</i>	Exporting an animated scene into VRML – each scene (animation frame) is an individual VRML file
<i>Parameters</i>	S1 – file path and name of the output VRML file
<i>Example</i>	O_OutputAction 2 0 ./scene.wrl
O_OutputAction 2 1	
<i>Description</i>	Exporting an animated scene into VRML – each scene (animation frame) is stored in a VRML header file and a geometry file
<i>Parameters</i>	S1 – file path and name of the VRML header file, S2 – file path and name of the VRML geometry file, I3 – use absolute path

	yes/no
<i>Example</i>	O_OutputAction 2 1 ./sceneHeader.wrl ./sceneGeom.wrl 0
O_OutputAction 2 2	
<i>Description</i>	Exporting an animated scene into VRML – each scene (animation frame) is stored in a VRML header file and a geometry file, the geometry is only being appended into a previously existing VRML header file
<i>Parameters</i>	S1 – file path and name of the VRML header file, S2 – file path and name of the VRML geometry file, I3 – use absolute path yes/no
<i>Example</i>	O_OutputAction 2 2 ./sceneHeader.wrl ./sceneGeom.wrl 0
O_OutputAction 2 3	
<i>Description</i>	Adding a static VRML geometry file into a sequence of VRML header files
<i>Parameters</i>	S1 – base of the indexed file name of the VRML header files, S2 – file path and name of the VRML geometry file being inserted, I3 – use absolute path yes/no
<i>Example</i>	O_OutputAction 2 3 ./sceneAnim.wrl ./sceneGeomStatic.wrl 0
O_OutputAction 2 4	
<i>Description</i>	Adding an animated VRML geometry into a sequence of VRML header files
<i>Parameters</i>	S1 – base of the indexed file name of the VRML header files, S2 – the base of the indexed file name VRML geometry file being inserted, I3 – use absolute path yes/no
<i>Example</i>	O_OutputAction 2 4 ./sceneAnim.wrl ./sceneGeomAnim.wrl 0
O_OutputAction 2 5	
<i>Description</i>	Adding animated cameras (viewpoints) into a sequence of VRML header files
<i>Parameters</i>	S1 – base of the indexed file name of the VRML header files
<i>Example</i>	O_OutputAction 2 5 ./sceneAnim.wrl
O_OutputAction 2 6	
<i>Description</i>	Adding animated lights into a sequence of VRML header files
<i>Parameters</i>	S1 – base of the indexed file name of the VRML header files, I2 – use absolute path yes/no
<i>Example</i>	O_OutputAction 2 6 ./sceneAnim.wrl
O_OutputAction 2 7	
<i>Description</i>	Duplicates the given header file and uses proper indexing for the created copies
<i>Parameters</i>	S1 – file path and name of the VRML file to be duplicated
<i>Example</i>	O_OutputAction 2 7 ./sceneAnim.wrl
O_OutputAction 3 0	
<i>Description</i>	Exporting an animated scene into PBRT – each scene (animation frame) is an individual PBRT file
<i>Parameters</i>	S1 – file path and name of the output PBRT file, I2 – output the rendering linux script yes/no
<i>Example</i>	O_OutputAction 3 0 ./sceneAnim.pbrt 1
O_OutputAction 3 1	

<i>Description</i>	Exporting an animated scene into PBRT – each scene (animation frame) is stored in a PBRT header file and a geometry file
<i>Parameters</i>	S1 – file path and name of the PBRT header file, S2 – file path and name of the PBRT geometry file, I3 – use absolute path yes/no, I4 – output the rendering linux script yes/no
<i>Example</i>	O_OutputAction 3 1 ./sceneHeader.pbrt ./sceneGeom.pbrt 0 1
O_OutputAction 3 2	
<i>Description</i>	Exporting an animated scene into PBRT – each scene (animation frame) is stored in a PBRT header file and a geometry file, the geometry is only being appended into a previously existing PBRT header file
<i>Parameters</i>	S1 – file path and name of the PBRT header file, S2 – file path and name of the PBRT geometry file, I3 – use absolute path yes/no, I4 – output the rendering linux script yes/no
<i>Example</i>	O_OutputAction 3 2 ./sceneHeader.pbrt ./sceneGeom.pbrt 0 1
O_OutputAction 3 3	
<i>Description</i>	Adding a static PBRT geometry file into a sequence of PBRT header files
<i>Parameters</i>	S1 – file path and name of the PBRT header file, S2 – file path and name of the PBRT geometry file, I3 – use absolute path yes/no
<i>Example</i>	O_OutputAction 3 3 ./sceneHeader.pbrt ./sceneGeom.pbrt 0
O_OutputAction 3 4	
<i>Description</i>	Adding an animated PBRT geometry into a sequence of PBRT header files
<i>Parameters</i>	S1 – file path and name of the PBRT header file, S2 – file path and name of the PBRT geometry file, I3 – use absolute path yes/no
<i>Example</i>	O_OutputAction 3 4 ./sceneHeader.pbrt ./sceneGeom.pbrt 0
O_OutputAction 3 5	
<i>Description</i>	Adding animated camera into a sequence of PBRT header files
<i>Parameters</i>	S1 – file path and name of the PBRT header file
<i>Example</i>	O_OutputAction 3 5 ./sceneHeader.pbrt
O_OutputAction 3 6	
<i>Description</i>	Adding animated lights into a sequence of PBRT header files
<i>Parameters</i>	S1 – file path and name of the PBRT header file, S2 – file path and name of the PBRT geometry file, I3 – use absolute path yes/no
<i>Example</i>	O_OutputAction 3 6 ./sceneHeader.pbrt ./sceneGeom.pbrt 0
O_OutputAction 3 7	
<i>Description</i>	Duplicates the given header file and uses proper indexing for the created copies
<i>Parameters</i>	S1 – file path and name of the VRML file to be duplicated, I2 – output the rendering linux script yes/no
<i>Example</i>	O_OutputAction 3 7 ./sceneHeader.pbrt 1
O_OutputAction 3 8	
<i>Description</i>	Adds an indexed output file name into a sequence of PBRT main file headers

<i>Parameters</i>	S1 – file path and name of the VRML file
<i>Example</i>	O_OutputAction 3 7 ./sceneHeader.pbrt
O_OutputAction 3 9	
<i>Description</i>	Updates the rendering information in a sequence of PBRT main files
<i>Parameters</i>	S1 – file path and name of the VRML file
<i>Example</i>	O_OutputAction 3 9 ./sceneHeader.pbrt

Appendix E – User Manual

This appendix contains serves as a user manual for the VRMLtoPBRT application. First, all components of the Graphical User Interface (GUI) of the application are described. Next, three scenarios are explained in detail, guiding the user through the application workflow.

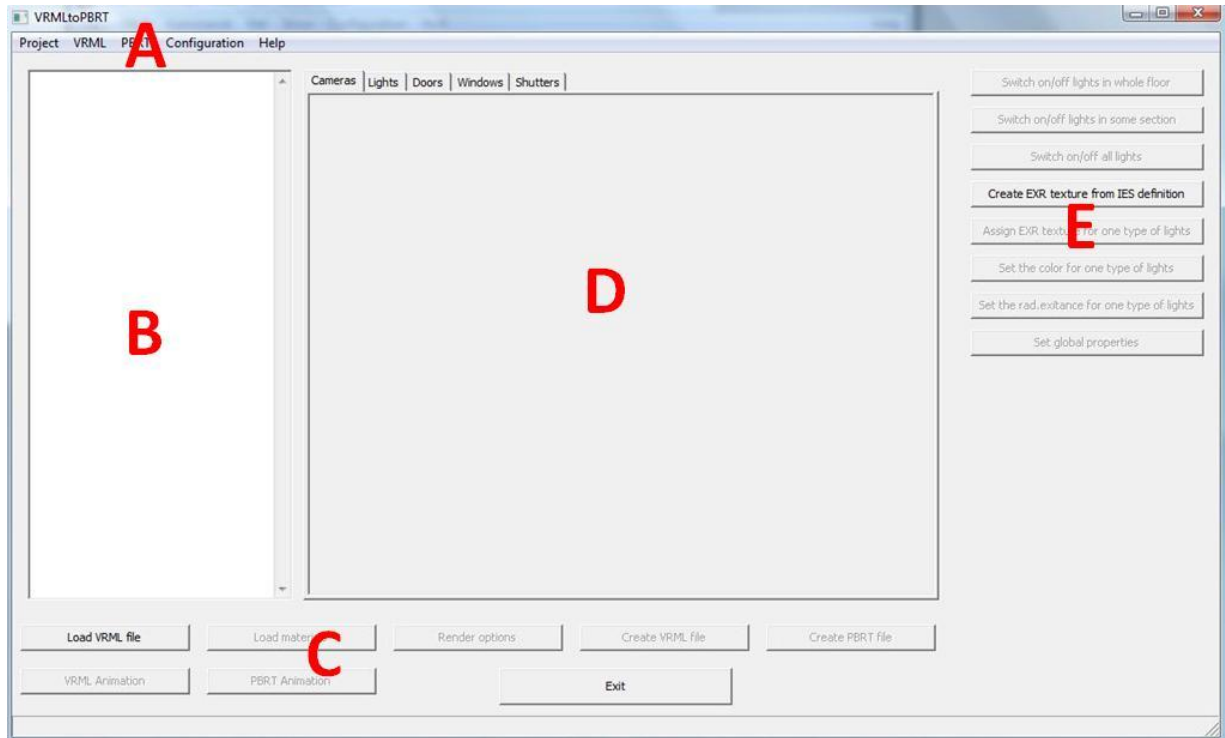


Fig. 57 Main panel of the VRMLtoPBRT application.

E.1 Application GUI

The graphical panel of the VRMLtoPBRT application is displayed in Fig. 57. This main panel provides access to all important parts of the application control. As denoted by red letters in Fig. 57, the panel can be divided into five main parts. Each part is described below in more detail.

E.1.1 Menu Options (A)

A detailed view of the menu options panel is provided in Fig. 58. The menu contains dialogs for opening and saving work projects, loading and creating VRML files, generating PBRT files, configuring the VRMLtoPBRT application and an application help. The project

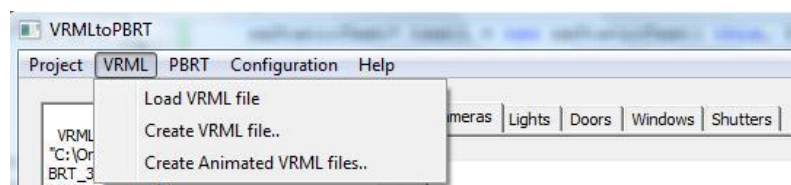


Fig. 58 Main options in the application GUI.

menu allows for loading both individual project and project lists. The configuration menu allows for selecting if the extended VRML format should be used. The help menu contains both information about the application and a brief help with a short description of the main functionalities of the GUI.

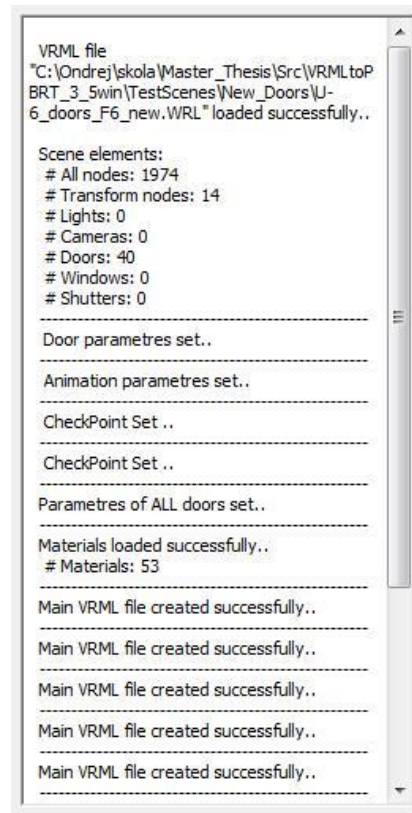


Fig. 59 Information panel of the GUI.

E.1.2 Information Panel (B)

The information panel is displayed in more detail in Fig. 59. The information panel shows a log of actions performed by the user.

E.1.3 Data Conversion Options (C)

Part C, depicted in more detail in Fig. 60, contains several buttons for data conversion of the MPII model:

- **Load VRML File:** Allows for loading the input VRML data into the application.
- **Load materials:** Button for loading the desired material files into the application.
- **Render options:** Opens a dialog for specifying the rendering options for the PBRT rendering engine.
- **Create VRML file:** Opens a dialog for producing VRML data for a static scene.
- **Create PBRT file:** Opens a dialog for producing PBRT data for a static scene.

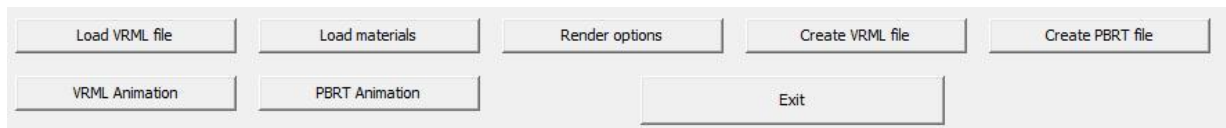


Fig. 60 Data conversion options.

- **VRML Animation:** Opens a dialog for producing VRML animation data for a dynamic scene.
- **PBRT Animation:** Opens a dialog for producing PBRT animation data for a dynamic scene.
- **Exit:** Exits the VRMLtoPBRT application.

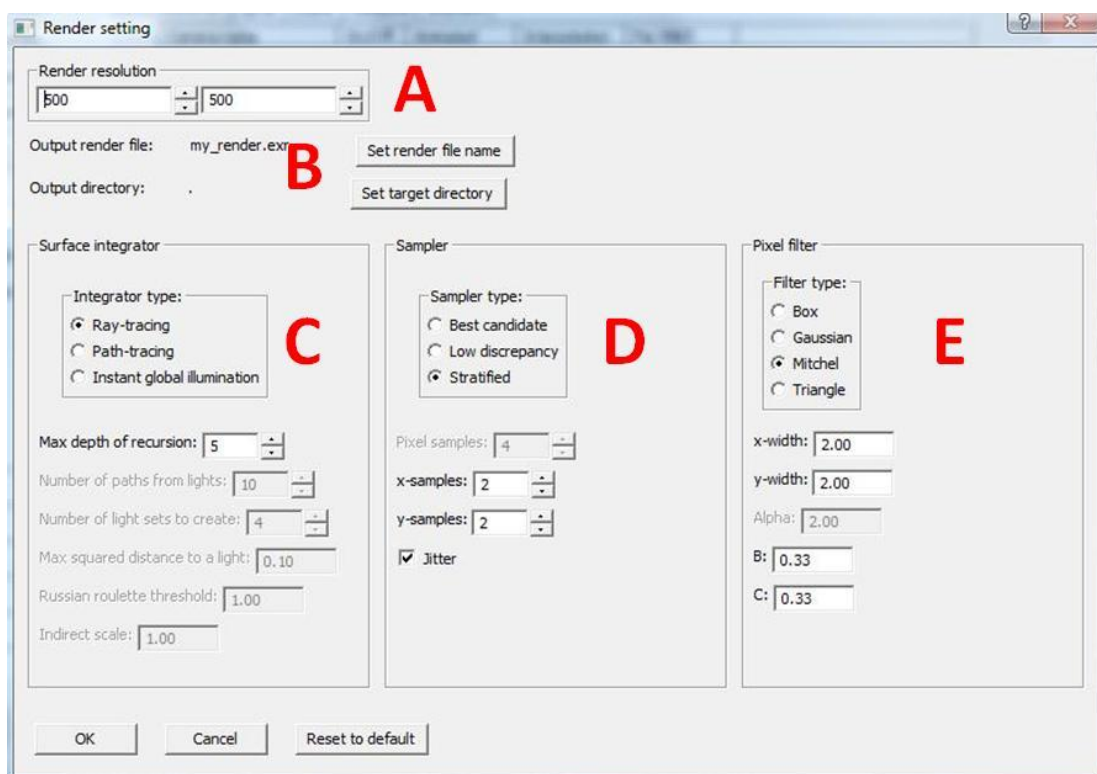


Fig. 61 Rendering options dialog.

E.1.3.1 Rendering Options

The rendering dialog is depicted in Fig. 61. It can be used for specifying the PBRT rendering options, which are then inserted into the PBRT output files. The dialog can be divided into five distinct parts:

- **Part A:** Specification of the resolution of the rendered image.
- **Part B:** Specification of the file name and the file path of the rendered image.
- **Part C:** This part of the dialog enables selection of one of the three supported PBRT surface integrators. The parameters of the selected surface integrator can be also selected here.

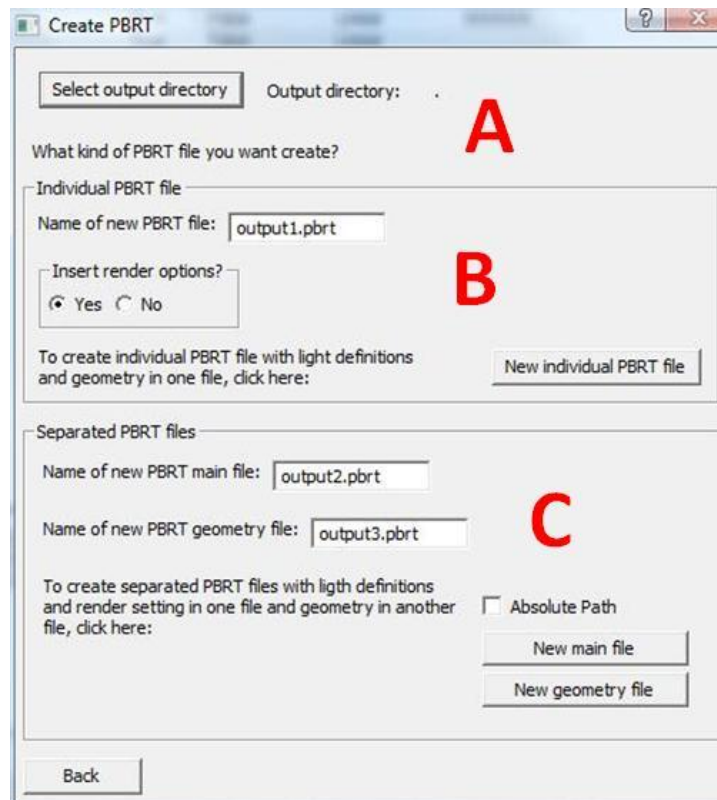


Fig. 62 Dialog for creating a PBRT output data for a static scene.

- **Part D:** The middle part of the dialog allows for selection of one of the supported PBRT samplers along with specification of its attributes.
- **Part E:** In the most right panel of the dialog, the pixel filter of for the PBRT rendering can be selected.

E.1.3.2 Create PBRT File

Fig. 62 shows the dialog for producing output PBRT data for a single static scene. The dialog can be decomposed into three main parts:

- **Part A:** Here the output director can be selected. The created files will be stored in the selected directory.
- **Part B:** The middle part of the dialog allows for producing an individual PBRT file. The generated individual PBRT file will contain the camera description, rendering definitions and lights declaration together with the scene geometry in a single PBRT file. The file name of the output PBRT can be specified here.
- **Part C:** This part of the dialog allows for producing separated PBRT files for the single static scene. The main file will contain render, camera and lights definitions. The geometry of the scene will be stored in the geometry file.

E.1.3.3 VRML Animation

Fig. 63 depicts the dialog for producing VRML data for an animated scene. The dialog can be divided into 4 parts as follows:

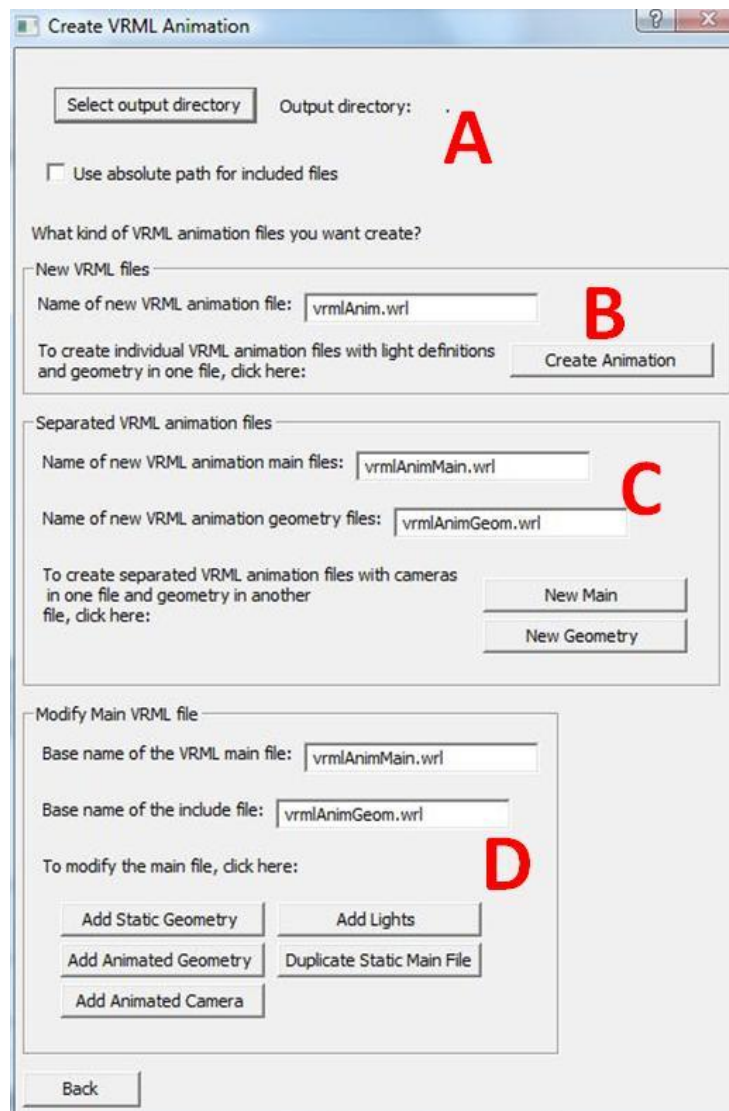


Fig. 63 Dialog for creating a VRML output data for an animated scene.

- **Part A:** Here the output director can be selected. The created files will be stored in the selected directory. In addition, a checkbox for specifying if absolute file path should be used during includes is located here.
- **Part B:** This part of the dialog allows for producing an animation in individual VRML files. The generated individual VRML files will contain the viewport description and lights declaration together with the scene geometry in a single file. The file name of the output VRML can be specified here.
- **Part C:** This part of the dialog allows for producing an animation in separated VRML files. The main files will contain viewport specification and lights definitions. The animated geometry of the scene will be stored in the geometry files. By using the „New Main“ option, new main files will be generated for each animation frame. By using the „New Geometry“ option, the produced animated VRML geometry files will only be included into pre-existing VRML main files. The main files must exists in the specified location, otherwise and error will be reported.

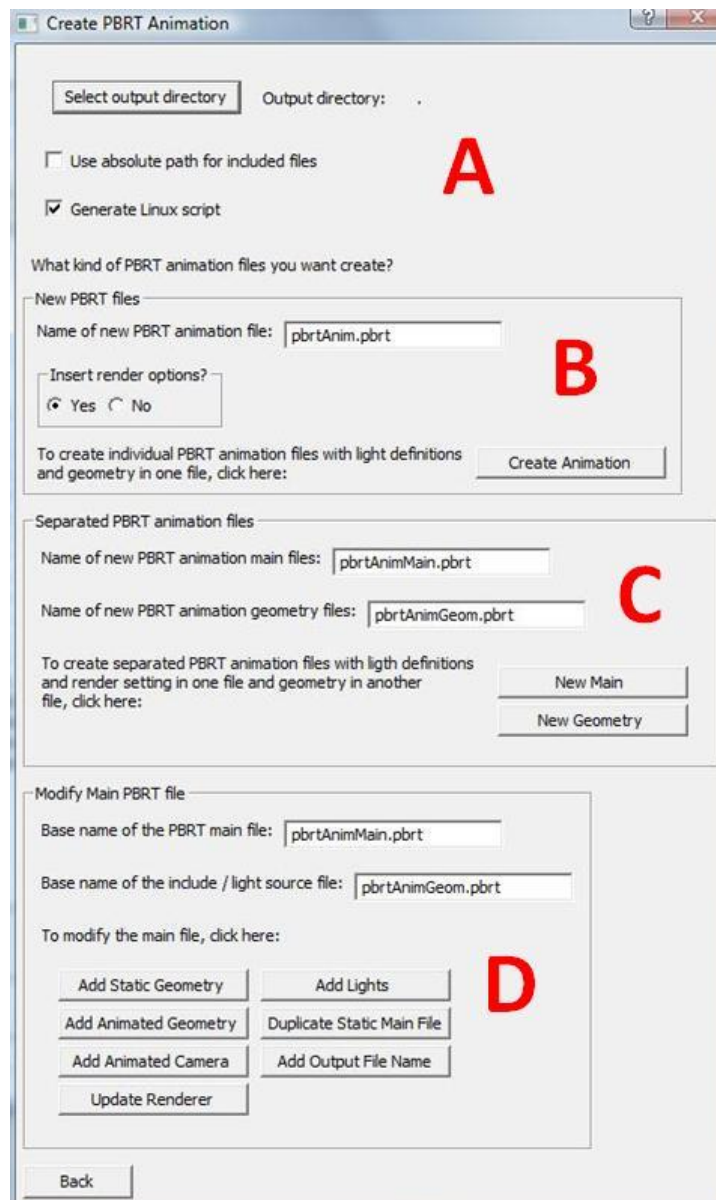


Fig. 64 Dialog for creating a PBRT output data for an animated scene.

- **Part D:** The bottom part of the dialog specifies additional functions for VRML animation processing. These include: adding static geometry files into animated VRML main files, adding animated geometry into animated VRML main files, adding an animated camera into animated VRML main files, adding lights into animated VRML main files and duplicating a static VRML main file.

E.1.3.4 PBRT Animation

Fig. 64 depicts the dialog for producing PBRT data for an animated scene. The dialog can be divided into 4 parts as follows:

- **Part A:** Here the output director can be selected. The created files will be stored in the selected directory. In addition, a checkbox for specifying if absolute file path should be used during includes is located here. Also a selection can be made if a Linux script of automatic rendering of the generated data should be produced.

- **Part B:** This part of the dialog allows for producing an animation in individual PBRT files. The generated individual PBRT files will contain the camera specification, rendering options and lights declaration together with the scene geometry in a single file. The file name of the output PBRT can be specified here.
- **Part C:** This part of the dialog allows for producing an animation in separated PBRT files. The main files will contain camera specification, rendering options and lights definitions. The animated geometry of the scene will be stored in the geometry files. By using the „New Main“ option, new main files will be generated for each animation frame. By using the „New Geometry“ option, the produced animated PBRT geometry files will only be included into pre-existing PBRT main files. The main files must exists in the specified location, otherwise and error will be reported.
- **Part D:** The bottom part of the dialog specifies additional functions for PBRT animation processing. These include: adding static geometry files into animated PBRT main files, adding animated geometry into animated PBRT main files, adding an animated camera into animated PBRT main files, adding lihghts into animated PBRT main files, updating the rendering options in a sequence of animated PBRT main files, updating the output file names in a sequence of animated PBRT main files and duplicating a static PBRT main file.

Cameras Lights Doors Windows Shutters								
Door name	Type	Floor	Section	Rotation	Handle	Animated	Interpolation	
DOOR_01	3	6	2	0	0	False	Linear	
DOOR_02	4	6	2	0	0	False	Linear	
DOOR_06	3	6	1	0	0	False	Linear	
DOOR_07	3	6	1	0	0	False	Linear	
DOOR_09	3	6	1	0	0	False	Linear	
DOOR_10	3	6	1	0	0	False	Linear	
DOOR_12	3	6	1	0	0	False	Linear	
DOOR_14	3	6	1	0	0	False	Linear	
DOOR_08	4	6	1	0	0	False	Linear	
DOOR_11	4	6	1	0	0	False	Linear	
DOOR_13	4	6	1	0	0	False	Linear	
DOOR_15	4	6	1	0	0	False	Linear	
DOOR_19	4	6	4	0	0	False	Linear	
DOOR_20	4	6	4	0	0	False	Linear	
DOOR_18	3	6	4	0	0	False	Linear	
DOOR_21	4	6	3	0	0	False	Linear	
DOOR_22	4	6	3	0	0	False	Linear	
DOOR_24	4	6	3	0	0	False	Linear	
DOOR_26	4	6	3	0	0	False	Linear	
DOOR_23	3	6	3	0	0	False	Linear	
DOOR_25	3	6	3	0	0	False	Linear	
DOOR_04	3	6	2	0	0	False	Linear	
DOOR_05	3	6	2	0	0	False	Linear	
DOOR_03	4	6	2	0	0	False	Linear	
DOOR_36	1	6	2	0	0	False	Linear	
DOOR_37	1	6	2	0	0	False	Linear	
DOOR_38	1	6	2	0	0	False	Linear	
DOOR_39	1	6	2	0	0	False	Linear	
DOOR_27	3	6	2	0	0	False	Linear	
DOOR_35	1	6	2	0	0	False	Linear	
DOOR_33	1	6	2	0	0	False	Linear	
DOOR_31	1	6	2	0	0	False	Linear	

Fig. 65 Object description panel.

E.1.4 Objects Descriptions (D)

Fig. 65 shows the objects descriptions panels displaying the information about the available door objects in the loaded VRML scene. By clicking at one of the tabs at the top of the panel the available cameras, lights, doors, windows or window shutter objects can be

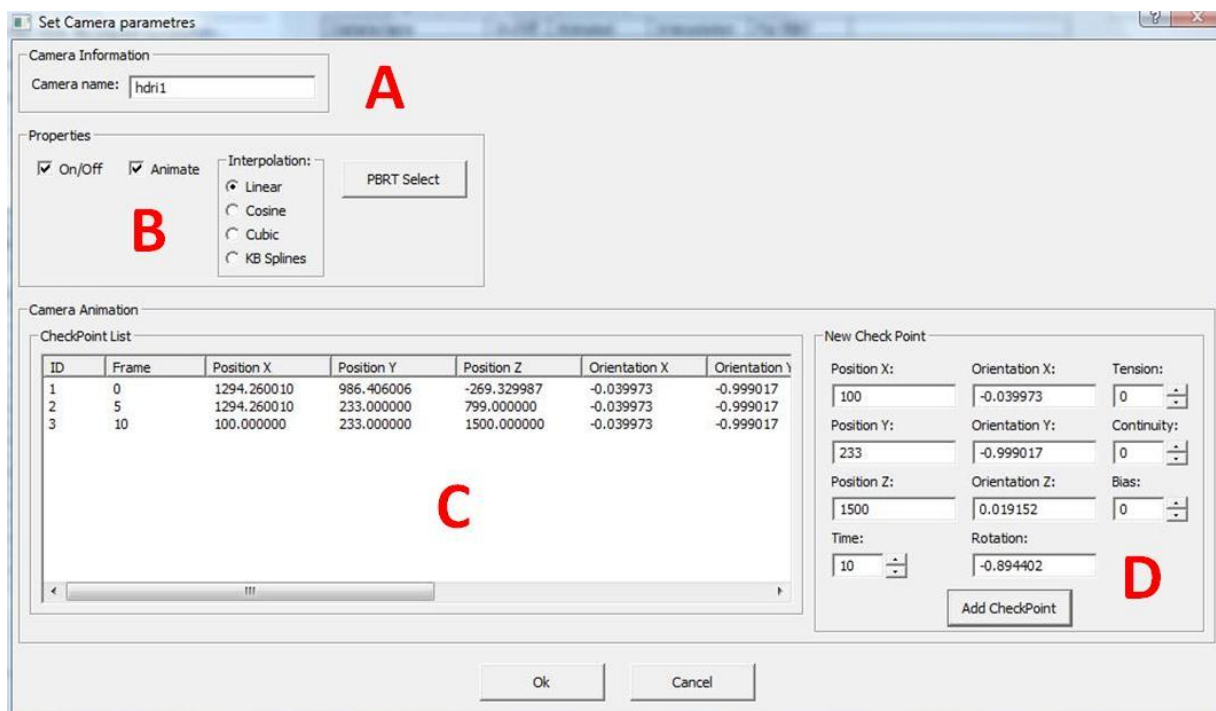


Fig. 66 Camera modification dialog.

viewed. Clicking at the selected object of interest an object modification dialog can be accessed. Bellow is a description of the camera and door modification dialogs. The other dialogs are similar to the presented ones.

E.1.4.1 Camera Modification Dialog

Fig. 66 depicts the dialog for modification of the camera objects. The dialog can be divided into four main parts:

- **Part A:** In the text box, the name of the selected camera is displayed.
- **Part B:** The part of the dialog contains controls some of the most important properties of the camera object. A camera can be turned on or off. Cameras that are off will not be included in the generated VRML files. Next, the animation flag can be set here. If the animation checkbox is set, the animation part of the dialog will be enabled. One of the four available interpolations techniques can be selected for camera animations. The „PBRT Select“ button determines if this camera should be used as the PBRT camera. Only one single camera is always selected for PBRT rendering.
- **Part C:** This panel displays the check-point list of the produced animation. Each record has a unique ID and contains description of its time and all its parameters.
- **Part D:** New animation check-point can be created here. Upon parameter selections, the check-point can be added into the animation by clicking at the „Add CheckPoint“ button.

E.1.4.2 Door Modification Dialog

Fig. 67 shows the dialog for modifying the properties of the recognized door objects. The dialogs for modification of windows and window shutters are very similar. Properties of

Set Door parameters

Door Information

Door name:

Type: Floor: Section:

Geometry Modification

Rotation: Handle Rate: ☒ Animate

Interpolation:
☒ Linear
☐ Cosine

Door Animation

CheckPoint List

ID	Frame	Rotation	Handle
1	0	0	0
2	3	50	45
3	7	90	100
4	10	30	0

New Check Point

Rotation:

Handle Rate:

Time:

Modify Multiple Objects

Fig. 67 Door modification dialog.

individual door objects as well as animation of a single or multiple door objects can be specified here. The dialog can be divided into five main parts:

- **Part A:** In the top part of the dialog the information about the selected door is displayed.
- **Part B:** This section of the dialog allows for modification of the parameters of the selected door object in a static scene. The rotation of the door as well as the rotation of the door handle can be specified here. Next, the animation flag can be set here. Setting the animation flag to true enables the bottom part of the dialog with door animation options. Finally, the interpolation method for the animation processing can be selected here.
- **Part C:** This panel displays the check-point list of the produced animation. Each record has a unique ID and contains description of its time and all its parameters.
- **Part D:** New animation check-point can be created here. Upon parameter selections, the check-point can be added into the animation by clicking at the „Add CheckPoint“ button.
- **Part E:** This part of the dialog contains selection of the application mode of the constructed object modification. By clicking at the „Ok“ button, the object

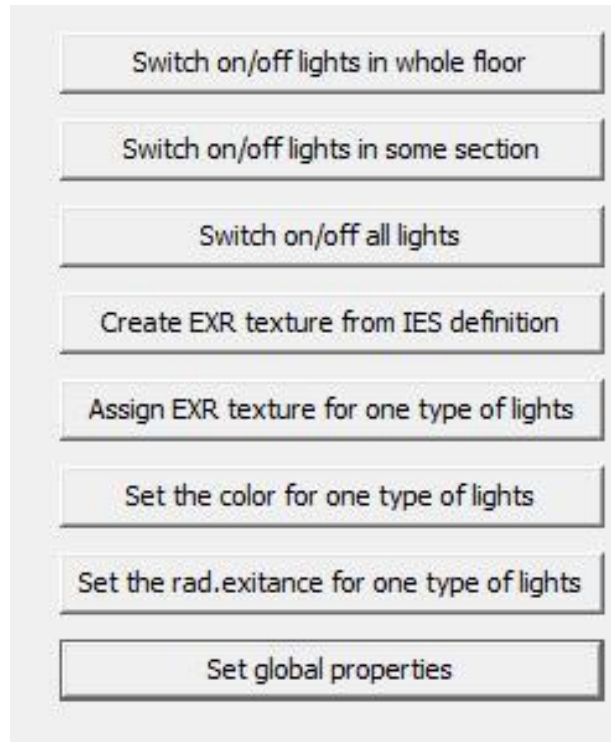


Fig. 68 Settings options.

modification will be only assigned to the currently selected door object. Clicking at the „Section“, „Floor“ or „All“ buttons will assign the object modification to all objects in the same section, on the same floor and in the whole dataset as the currently selected door object, respectively.

E.1.5 Settings (E)

This panel on the right side of the GUI offeres additional functionalities. Eight buttons are located here with the following functions:

- **Switch on/off lights in whole floor:** Opens a dialog for determining the on/off mode of all lights on a certain floor.
- **Switch on/off lights in some section:** Opens a dialog for determininig the on/off mode of all lights in a certain section.
- **Switch on/off all lights:** Opens a dialog for determining the on/off mode of all lights in the model.
- **Create EXR texture from IES definition:** Opens dialog for creating an EXR texture based on a selected IES light definition.
- **Assign EXR texture for one type of lights:** Opens a dialog for assigning a specific EXR texture to a selected type of lights.
- **Set the color for one type of lights:** Opens a dialog for specifying the color of a certain type of lights.

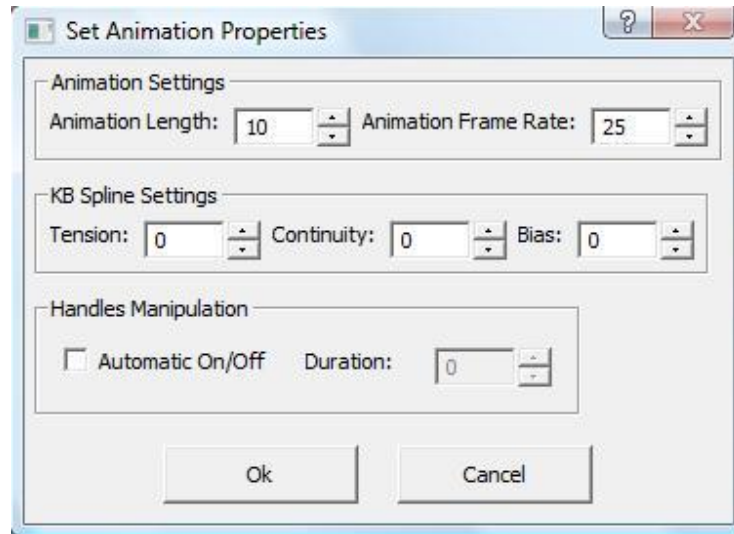


Fig. 69 Global animation settings.

- **Set global properties:** Opens a dialog for setting the global properties of the animation process, e.g. animation length and framerate.

E.1.5.1 Global Properties.

The global properties dialog is depicted in Fig. 69. In the top part of the dialog, the animation length and the animation frame rate can be specified. Specifying a non-zero animation length will enable animation options in the rest of the VRMLtoPBRT application.

The part of the dialog, the KB spline global settings can be selected. The selected parameters will be applied as default values to any KB spline interpolation nodes used in the animation of cameras. These values can be overwritten by specifying a local parameteres of specific animation nodes.

In the bottom part of the dialog, the automatic handle manipulation can be turned on/off. Turning this feature on, will disable the possibility to animate door and window handles, but these will be animated automatically based on the opening rate of the object. The duration of the automatic handle manipulation can also be specified here.

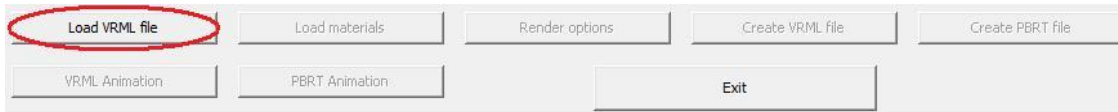
E.2 User Guide

Three exemplary scenarios were created to cover the main functionalities of the developed application. In the first scenario, a PBRT animation data are prepared. The performed steps are logged into a project file. In the second scenario, the PBRT animation data are modified using the application GUI. In the last scenario, the console version of the application is used to reproduce the data conversion from the first scenario using the created project file.

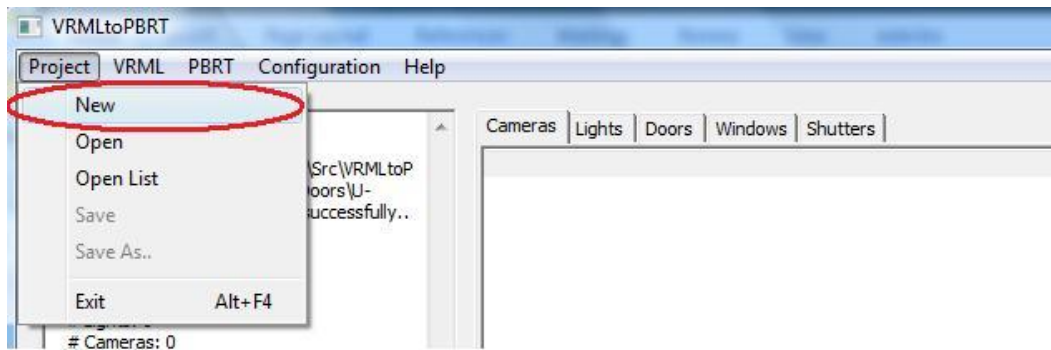
E.2.1 Scenario 1

In this scenario input VRML file containing door objects is first loaded into the application. Next, simple geometry modifications are applied to the door objects. After the animation properties are selected, PBRT animation data are produced. The workflow is logged into a project text file.

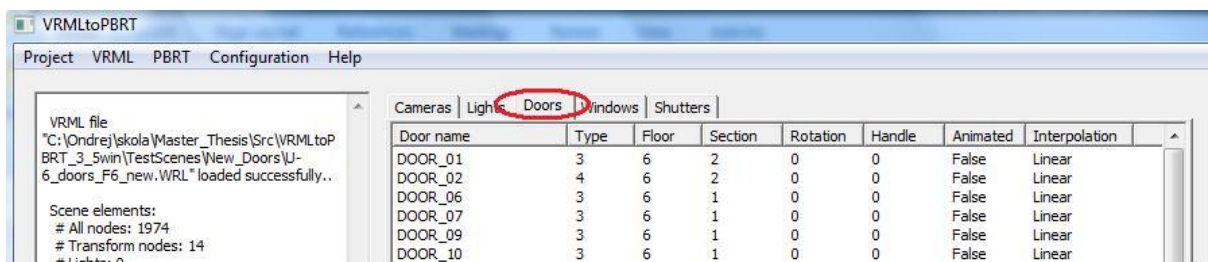
- **Step 1:** Load the VRML input data by clicking on the „Load VRML file“ button.



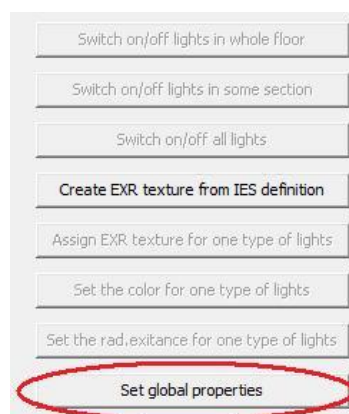
- **Step 2:** Create a new project log by selecting the Project/New option from the application menu.



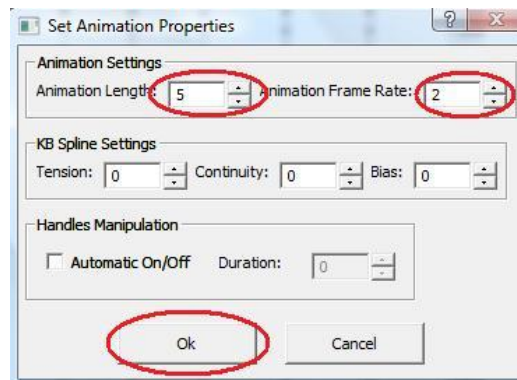
- **Step 3:** Switch to the information tab for recognized door objects.



- **Step 4:** Open the „Global Properties“ dialog by clicking on the „Set global properties“ button.



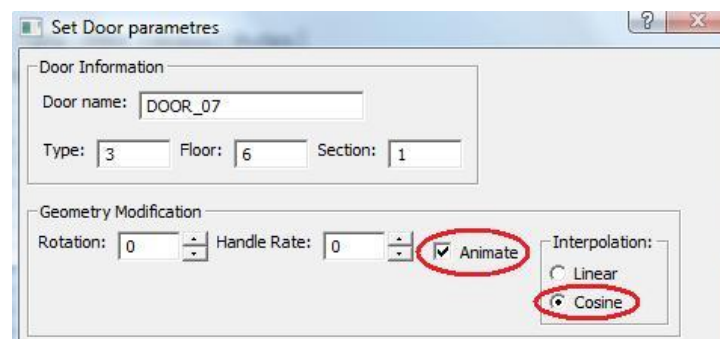
- **Step 5:** Specify the animation length and its framerate in the opened dialog. Click „Ok“.



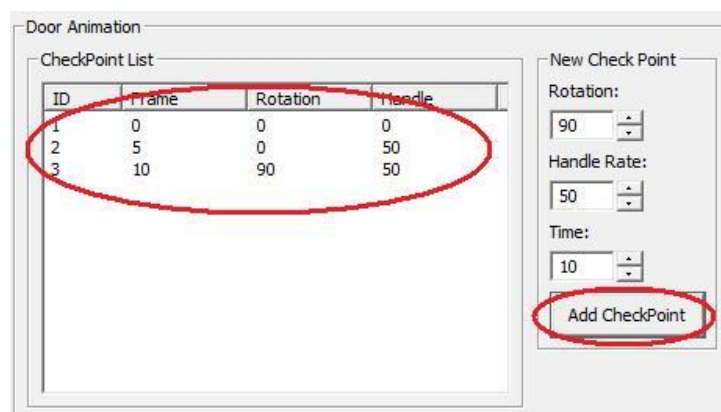
- **Step 6:** Open the object modification dialog by double-clicking on the selected door object.

Cameras Lights Doors Windows Shutters									
Door name	Type	Floor	Section	Rotation	Handle	Animated	Interpolation		
DOOR_01	3	6	2	0	0	False	Linear		
DOOR_02	4	6	2	0	0	False	Linear		
DOOR_06	3	6	1	0	0	False	Linear		
DOOR_07	3	6	1	0	0	False	Linear		
DOOR_09	3	6	1	0	0	False	Linear		
DOOR_10	3	6	1	0	0	False	Linear		
DOOR_12	3	6	1	0	0	False	Linear		
DOOR_14	3	6	1	0	0	False	Linear		

- **Step 7:** Enable object animations by checking the „Animate“ checkbox.
- **Step 8:** Select the desired interpolation method.



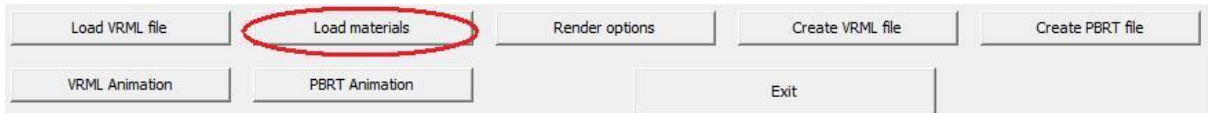
- **Step 9:** Create a series of animation checkpoints and add them to the animation checkpoint list of the selected door object.



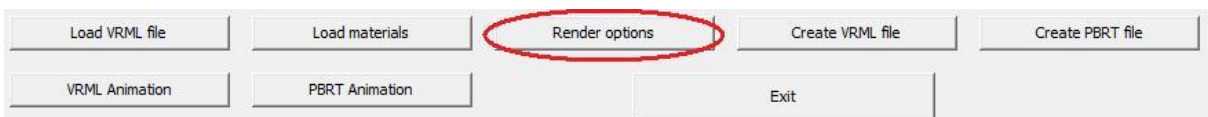
- **Step 10:** Apply the modifications (the create animation list) to the current door object by clicking on the „Ok“ button.



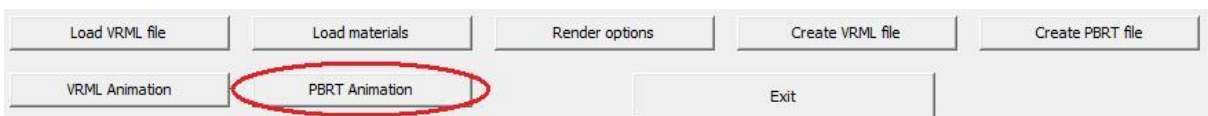
- **Step 11:** Load scene materials by clicking on the „Load materials“ button.



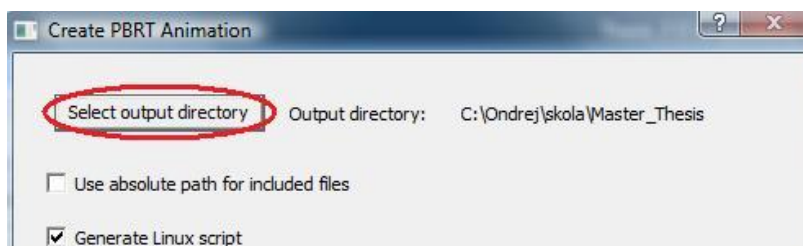
- **Step 12:** Click on the „Renderer options“ button to open the renderer settings dialog and select the desired PBRT rendering attributes.



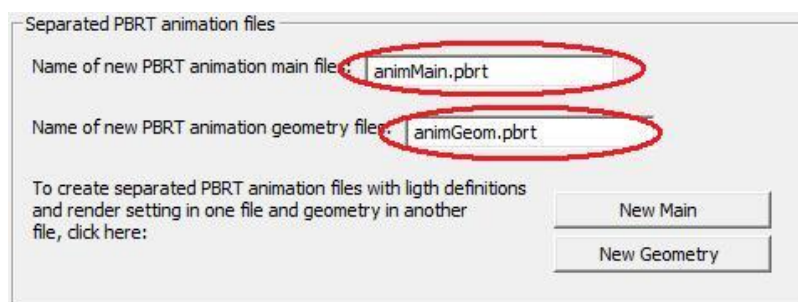
- **Step 13:** Open the dialog for creating the PBRT animation data by clicking on the „PBRT animation“ button.



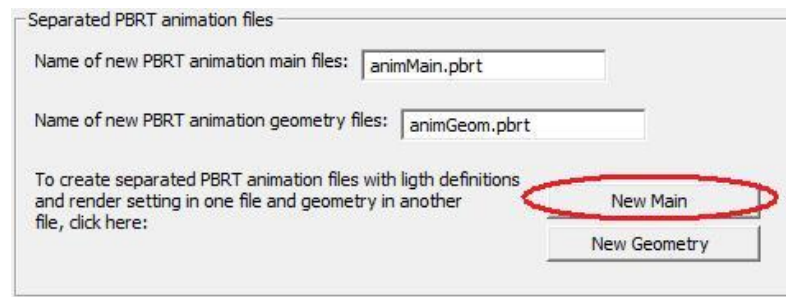
- **Step 14:** Select the output directory.



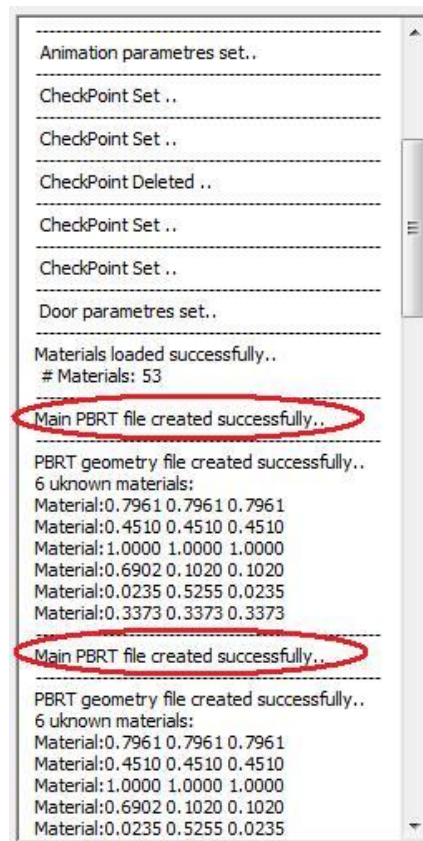
- **Step 15:** Select the names of the PBRT main and geometry files.



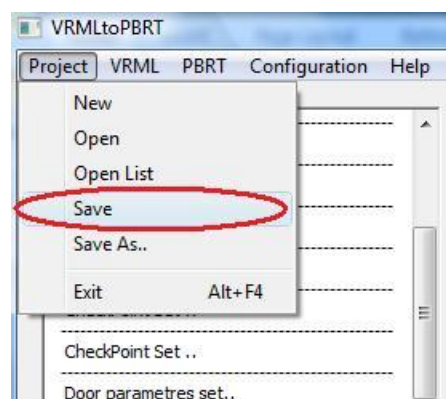
- **Step 16:** Click on the „New Main“ button to generate the desired data.



- **Step 17:** The succesfull completion of the PBRT data generation can be checked in the information panel.



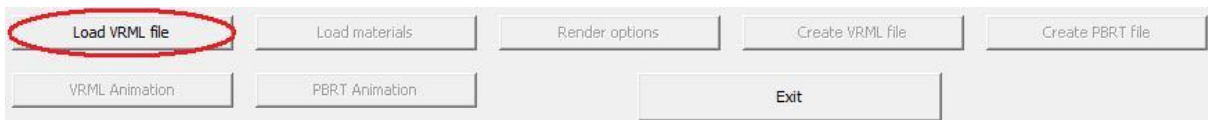
- **Step 18:** Store the performed actions in the project file (specified in **Step 2**) by clicking on Project/Save option from the application menu.



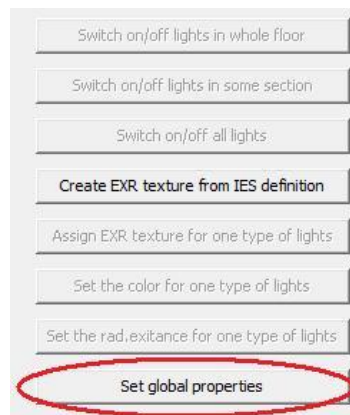
E.2.2 Scenario 2

In this tutorial the PBRT animation data from the previous scenario is modified using the VRMLtoPBRT application. First, the user loads an input file with camera definitions. A desired camera is selected for PBRT rendering and its animation is specified. The selected camera is inserted into the previously created PBRT main files.

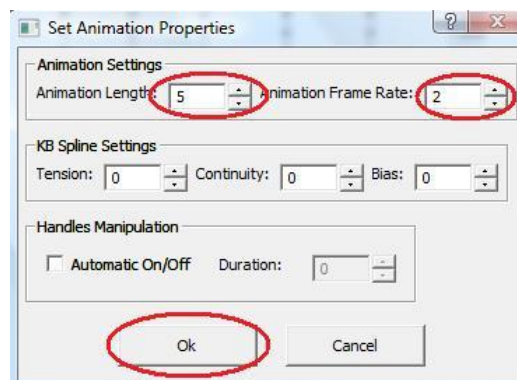
- **Step 1:** Load the VRML input data with the camera definitions by clicking on the „Load VRML file“ button.



- **Step 2:** Open the „Global Properties“ dialog by clicking on the „Set global properties“ button.



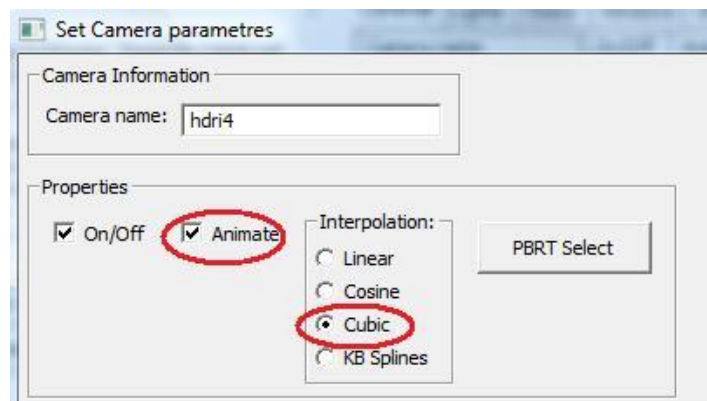
- **Step 3:** Specify the animation length and its framerate in the opened dialog. Set the same animation length and the same frame rate as it was used in Scenario 1. Click „Ok“.



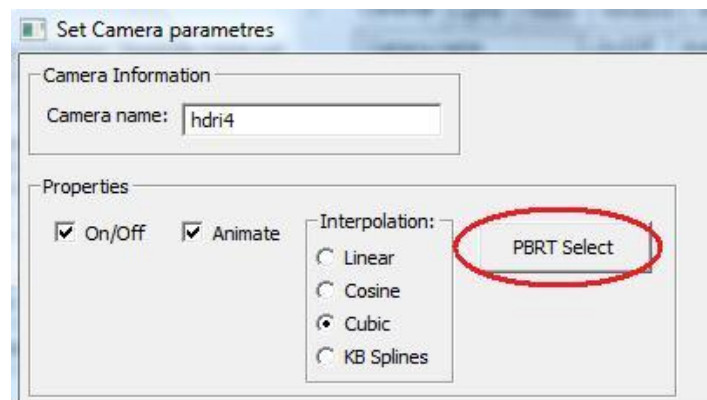
- **Step 4:** Open the object modification dialog by double-clicking on the desired camera object.

Cameras	Lights	Doors	Windows	Shutters
Camera name	On/Off	Animated	Interpolation	For PBRT
Hdri_21	True	False	Linear	XXXXXXX
hdri1	True	False	Linear	
hdri2	True	False	Linear	
hdri3	True	False	Linear	
hdri4	True	False	Linear	
hdri5	True	False	Linear	
hdri6	True	False	Linear	
hdri8	True	False	Linear	
Hdri_10	True	False	Linear	
Hdri_11	True	False	Linear	
Hdri_12	True	False	Linear	
Hdri_13	True	False	Linear	

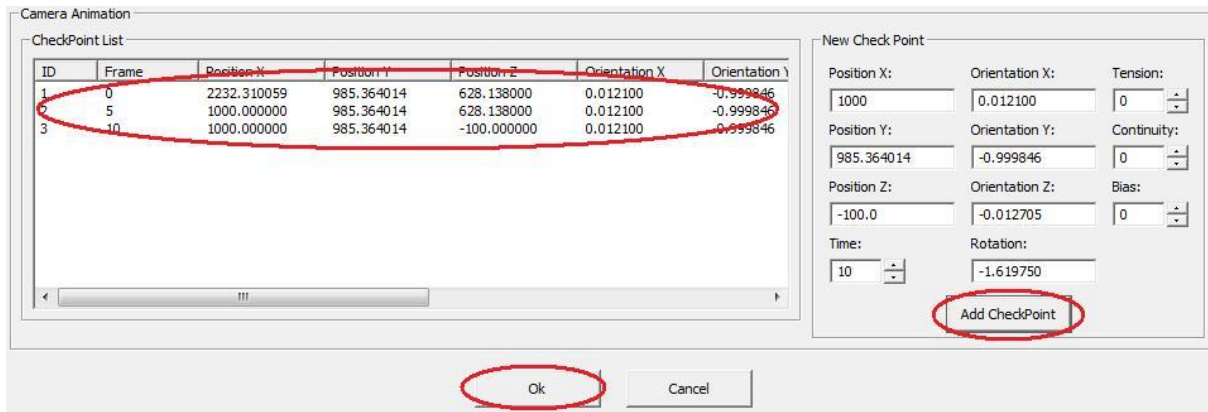
- **Step 5:** Enable object animations by checking the „Animate“ checkbox.
- **Step 6:** Select the desired interpolation method.



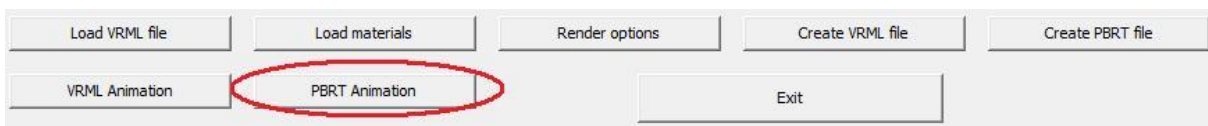
- **Step 7:** Click on the „PBRT Select“ button to choose this camera for PBRT rendering.



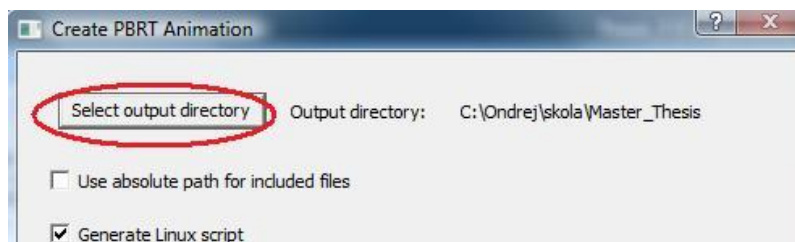
- **Step 8:** Create a series of animation checkpoints and add them to the animation checkpoint list of the selected camera.
- **Step 9:** Apply the modifications (the create animation list) to the current camera object by clicking on the „Ok“ button.



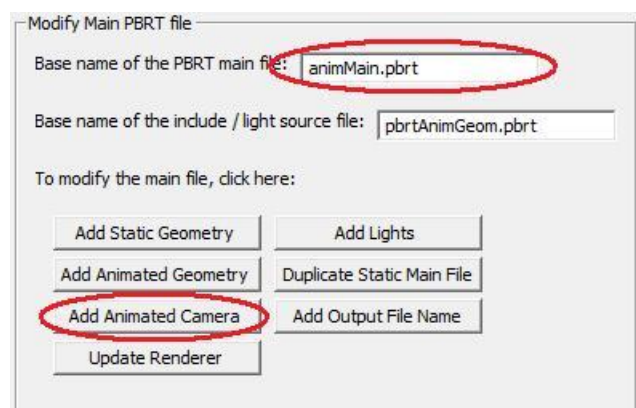
- **Step 10:** Open the dialog for creating the PBRT animation data by clicking on the „PBRT animation“ button.



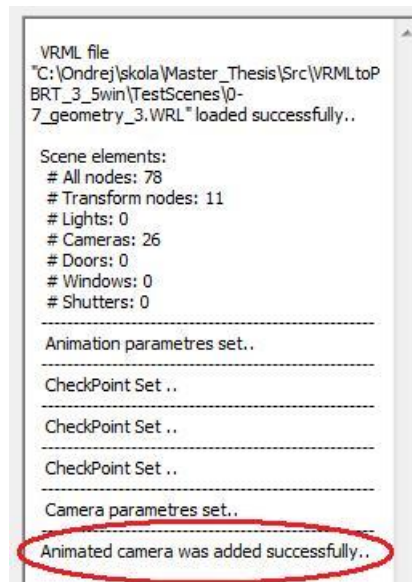
- **Step 11:** Select the output directory where the PBRT animation data from Scenario 1 were stored.



- **Step 12:** In the „Modify Main PBRT file“ section of the enter the base name of the PBRT main files generated in Scenario 1.
- **Step 13:** Click on the „Add Animated Camera“ button to add the animated camera to the previously generated PBRT animation files.



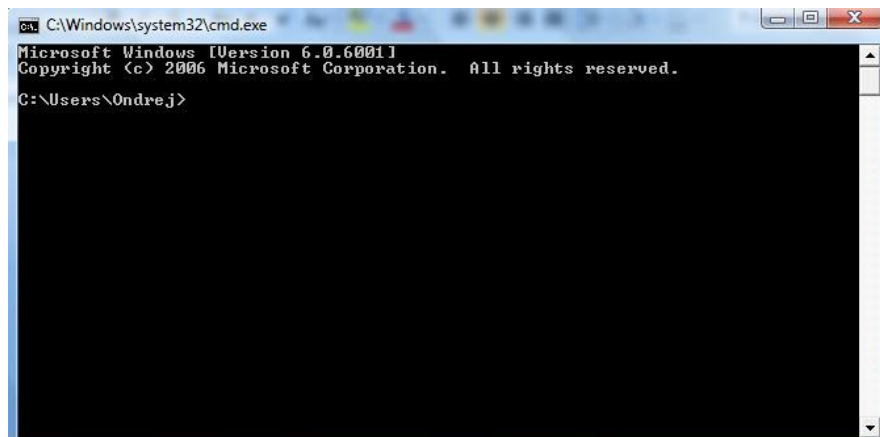
- **Step 14:** The succesfull completion of the PBRT data generation can be checked in the information panel.



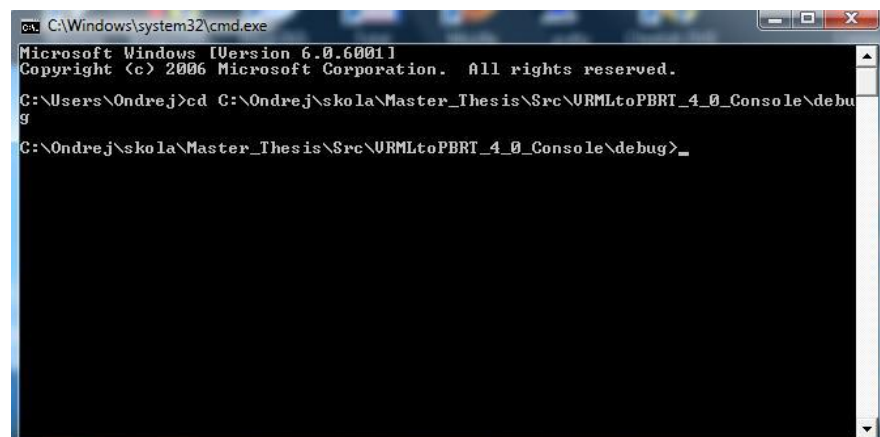
E.2.3 Scenario 3

In this simple scenario, the project file generated in the scenario 1 is processed using the console version of the VRMLtoPBRT application.

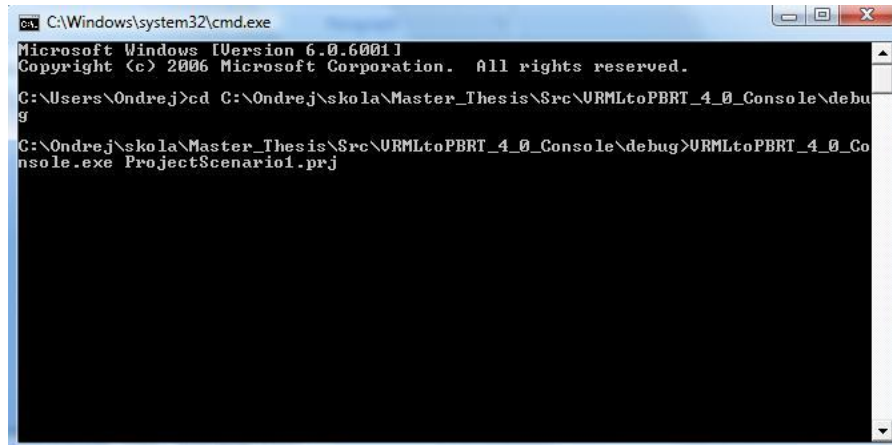
- **Step 1:** Open the command prompt window.



- **Step 2:** Move to the directory with the executable of the console version of the VRMLtoPBRT application.



- **Step 3:** Run the executable and supply the project file (or project list file) as a paramter.



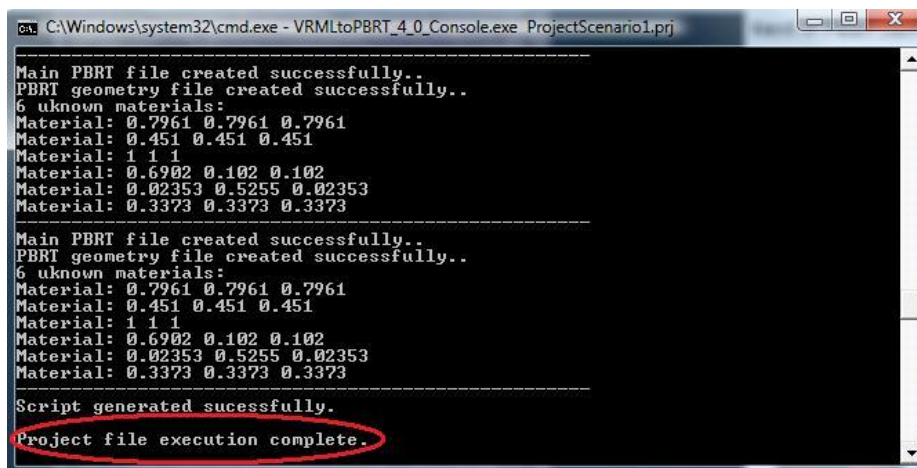
```

C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

C:\Users\Ondrej>cd C:\Ondrej\skola\Master_Thesis\Src\VRMLtoPBRT_4_0_Console\debug
C:\Ondrej\skola\Master_Thesis\Src\VRMLtoPBRT_4_0_Console\debug>VRMLtoPBRT_4_0_Console.exe ProjectScenario1.prj

```

- **Step 4:** The succesfull completion of the PBRT data generation can be checked by viewing the information messages printed into the command prompt window.



```

C:\Windows\system32\cmd.exe - VRMLtoPBRT_4_0_Console.exe ProjectScenario1.prj
-----
Main PBRT file created successfully..
PBRT geometry file created successfully..
6 unknown materials:
Material: 0.7961 0.7961 0.7961
Material: 0.451 0.451 0.451
Material: 1 1 1
Material: 0.6902 0.102 0.102
Material: 0.02353 0.5255 0.02353
Material: 0.3373 0.3373 0.3373
-----
Main PBRT file created successfully..
PBRT geometry file created successfully..
6 unknown materials:
Material: 0.7961 0.7961 0.7961
Material: 0.451 0.451 0.451
Material: 1 1 1
Material: 0.6902 0.102 0.102
Material: 0.02353 0.5255 0.02353
Material: 0.3373 0.3373 0.3373
-----
Script generated sucessfully.
Project file execution complete.

```

Appendix F – DVD Content

Application – Application data files.

Doc – Documentation of application source codes generated by Doxygen.

VS2005 – Application projects for Microsoft Visual Studio 2005.

Win32 – Executables of the application compiled for Windows OS.

Linux – Executables of the application compiled for Linux OS.

Model – MPII data model.

3DSMax – MPII data model for 3D Studio Max.

Lights_IES – IES definitions of lights.

Materials – Material files for the MPII model.

PBRT – Converted MPII model in the PBRT format.

VRML – Converted MPII model in the VRML format .

Results – Rendered results.

Animation1_StaticHall – Rendering files for animation 1.

Animation2_DynamicHall - Rendering files for animation 2.

Animation3_IntoAtrium – Rendering files for animation 3.

Animation4_Entrance – Rendering files for animation 4.

Animation5_Roundel – Rendering files for animation 5.

Text – Text of the Thesis.

Figures – Figures used in the thesis.

Thesis – Thesis text in different formats.

Recourses – Other resources.