České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Daniya Unembayeva**

Studijní program: Softwarové technologie a management
Obor: Web a multimedia

Název tématu: **Exportní modul pro Autodesk Maya do formátu X3D / X3Dom**

Pokyny pro vypracování:

Implementujte exportní plugin pro 3D modelář Autodesk Maya do formátu X3D / X3Dom. Soustřeďte se na podmnoužinu X3D, kterou je možno využívat v X3Dom. Základní objekty (viz specifikace X3D) exportujte jako odpovídající uzly X3D. S vedoucím vyberte podskupinu uzlů vhodných pro export, minimálně to budou uzly geometrické a uzly popisující vzhled objektů, kamery, světla, interpolátory základních transformačních vlastností. Umožněte zadefinování numerické přesnosti exportu a počtu vrcholů v indexovaných uzlech.

Otestujte na scénách konzultovaných s vedoucím práce.

Vytvořte HTML stránku na které budete prezentovat a distribuovat vámi vytvořený plugin včetně ukázek exportu.

Seznam odborné literatury:

http://www.x3dom.org/
BEHR J., ESCHLER P., JUNG Y., ZÖLLNER M.: X3DOM: a DOM-based HTML5/X3D integration model. In Proc. Web3D (2009), pp. 127–135

Vedoucí: Ing. David Sedláček, Ph.D.

Platnost zadání: do konce zimního semestru 2017/2018

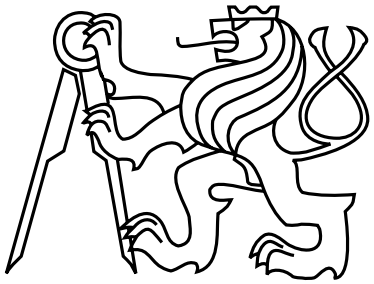
L.S.

prof. Ing. Jiří Žára, CSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 1. 3. 2016

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction



Bachelor's Project

X3D / X3DOM export plug-in for Autodesk Maya

Daniya Unembayeva

Supervisor: Ing. David Sedláček, Ph.D.

Study Programme: Softwarové technologie a management, Bakalářský

Field of Study: Web a multimedia

January 10, 2017

Aknowledgements

I take this opportunity to express gratitude to Ing. David Sedláček, my supervisor, for his help and valuable guidance. I also thank my family for their support and attention.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on January 10, 2017 .....

Abstract

The purpose of this bachelor's thesis is to create an X3D/X3DOM plug-in for Autodesk Maya. Basic objects will be exported to X3D/X3DOM by using corresponding nodes, more complex objects – by using a specific tags. Also there will be an opportunity to export not only a single object, but the whole scene, including lights, cameras, animation and information about the materials and textures. The plug-in will also allow to define the needed precision of numeric values written in the nodes' fields.

Key words
export plug-in, X3D, X3DOM

Abstrakt

Předmětem této bakalářské práce je vytvoření exportního modulu pro Autodesk Maya do formátu X3D/X3DOM. Základní objekty budou exportovány do X3D/X3DOM pomocí odpovídajících uzlů, složitější objekty - pomocí specifické značky. Také bude umožněno exportovat nejen jeden objekt, ale i celou scénu, včetně světel, kamer, animace a informace o materiálů a texturách. Exportní modul také umožní zadefinovat potřebnou numerickou přesnost hodnot napsaných do polí uzlů.

Klíčová slova
exportní modul, X3D, X3DOM

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Technological progress had a big effect on every human's life all around the world. Nowadays development in field of Computer Science reached such level that almost every day activity is in a some way or another connected with using the gifts of Computer Science. Good example of how Computer Science is integrating within human life is Computer Graphics. Advancement in field of Computer Graphics had a big impact on quality of every industry that is somehow connected with delivering visual experience. The most obvious demonstration of it will be the difference in quality of products in area of filming industry 20 years ago and in present times.

With internet spreading across the world and connection speed increasement Computer Graphics are becoming important part of multimedia web experience. Firstly as static 2D images, gif and flash animations rapidly developed into full 3D experience on the web. Rise of WebGL and HTML5 Canvas technologies led to better performance of 3D graphics for the web. However, due to the differences between notations of common 3D graphics' formats and xHTML notation there isn't much space for interactivity apart from simply embedding 3D graphics on the web page.

Fortunately there are 3D formats that are more native for the web. X3D 2.2 that is successor to VRML is a perfect example of 3D format that is web oriented because of it XML-like notation.

The goal of this bachelor's thesis is to create a plug-in for a popular 3D modeling software package Autodesk Maya 2.1 that will allow exporting graphical data created in this modeling software to X3D format for the later use on the web. Also considering the fact that modern browsers do not natively support X3D standard and the only implementation of X3D standard for web at this moment is .js library X3DOM, this plug-in will be capable of exporting graphical data directly to web file. This file will contain X3D data and plugged X3DOM library which will allow to immediately observe the result.

The link to the web page that contains information about this project can be found in **Appendix** D.

1.1 Organization of the Thesis

Chapter 2 contains the necessary theoretical background for a better understanding of Autodesk Maya software, X3D standard and X3DOM.

In Chapter 3 I describe design of plug-in software architecture as well as the design of export window for Autodesk Maya.

Chapter 4 is related to specific software solutions that were created during implementation process.

Chapter 5 is a showcase of the main capabilities of the plug-in. This chapter is made in form of functional and usability testing.

Finally Chapter 6 will conclude the achievements of the project and what the future work can be.

Chapter 2

Theoretical Background

2.1 Autodesk Maya

Autodesk Maya (shortened to Maya) is among most used software packages for making 3D graphics. Maya toolkit allows creating 3D models, texturing, animating objects, rendering video sequences, building bone structures for animation purposes. Single instance of work session in Maya's virtual workspace called *Scene*. There are various formats in which Maya *Scene* could be converted depending on the needs of every particular case. The Figure 2.1 illustrates a scene that was modelled in Autodesk Maya.

Figure 2.1: A 3D scene modelled in Maya [10].

Customization is one of the most remarkable feature of this 3D modeling software package. You can modify Maya in 4 ways: using

- MEL (Maya Embedded Language)
- Python

- Maya Python API
- C++ API

MEL is used for creating, editing, deleting Maya's GUI elements and for customizing Maya's changes per project. Python is good at providing an interface to the Maya commands. The Maya Python API allows writing scripts using the Python language. The Maya C++ API provides internal access to Maya and is used for implementing plug-ins and console applications to work with Maya. Plug-ins can be built in two ways: using Maya Python API or using Maya C++ API. In the first case the plug-in will be loaded to Maya as a script. So for my project I chose the Maya C++ API as I am going to do export a scene to an external file.

2.1.1 Maya system

According to David A. D. Gould [9] the Maya system is divided into three main parts (see Figure 2.2). While working in Maya user interacts with Maya GUI. But on the background all the user-to-software interactions are being executed by MEL commands. Before being executed these commands are sent to the Command Engine where they are interpreted and only then can be executed. Most of the MEL commands operate on Dependency Graph (DG). It defines data, data's structure in the scene and the data processing method.

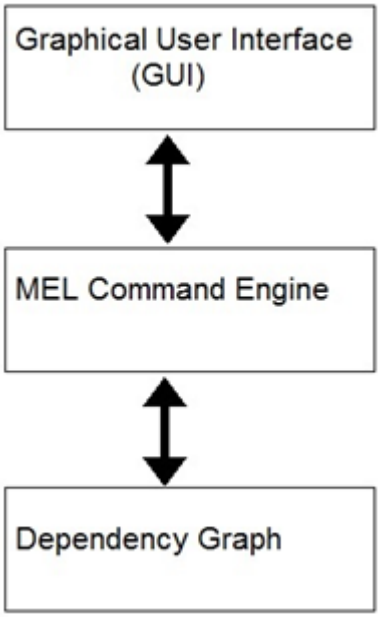


Figure 2.2: Maya system [9].

2.1.2 Dependency graph

Maya's core was implemented according to the data flow paradigm that represents applications as a directed graph with a set of nodes. The core is incarnated in the Dependency graph which represents the collection of nodes. The data and the operations are transferred through the connected attributes. The connection between the nodes is represented in Maya's Hypergraph window. Every task is completed by connecting nodes. The data from the first node is passed to the input of the next node then processed and sent to following. So the data are passed from the first to the last node in the nodes' network. Every node is meant to do specific operation. In case of the need to perform any complex modifications Maya will create a network consisting of other simple nodes.

The DG in Maya describes the whole scene, which may include not only the models themselves but the textures, lights and animation as well as a network of the connected nodes (see Figure 2.3). This network may be cyclic.

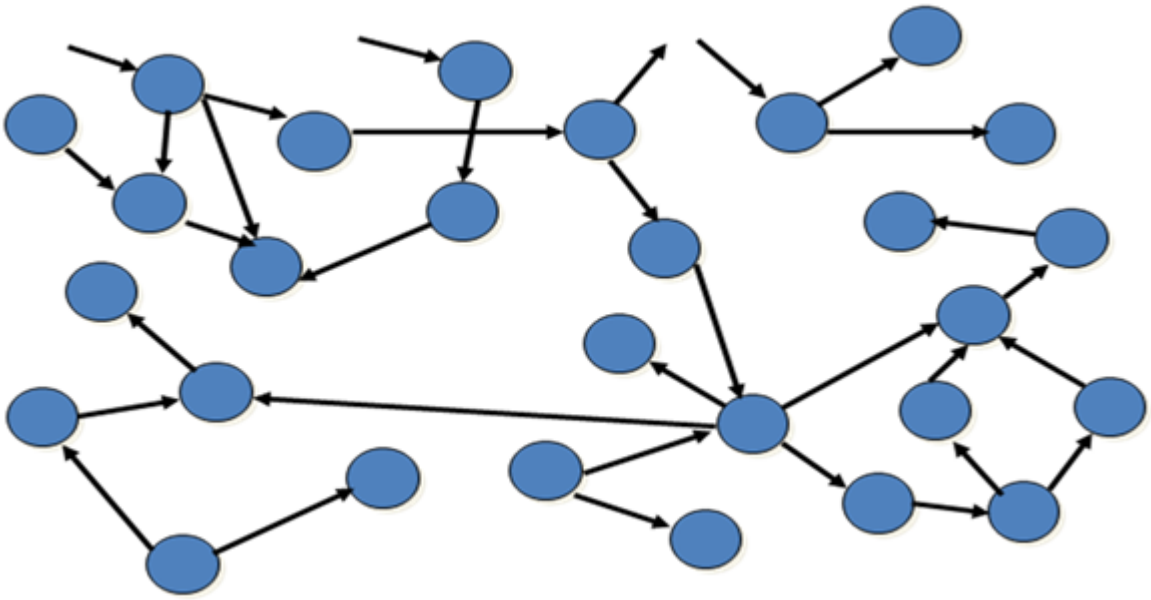


Figure 2.3: An example of the DG network.

2.1.3 Directed acyclic graph

Whenever a user wants to give a logical structure to some nodes, he is faced with the problem of defining a parent-child relationship between some objects. As the name implies, the connections of the DAG's elements, unlike the Dependency graph's nodes, cannot be cyclic and always represent the specific type of the relationship that is solving this issue.

Node in DAG cannot be both a parent and a child of itself. The DAG nodes extends functionality of DG nodes.

Maya Hypergraph can show the nodes either in their DAG „top-down“ hierarchical form (*Scene Hierarchy*) or as the Dependency graph nodes (*Input and Output Connections*).

DAG has a tree structure so that each DAG node is a leaf of the tree. [9] DAG paths are usually used in order to reference another node. A DAG path is a precise description of how to get to the node from the root by traversing down through the tree's nodes.

2.1.4 Nodes

Every node is set of inputs and outputs that depends on the outputs from the previous node in a chain and has attributes, plugs, data blocks, and data handlers.

An attribute of a node is a certain area containing specific information. Each attribute has a name, a structure (simple, array, compound, compound array), properties and a type of data being stored. It allows to define parent-child relationship as well.

A plug is pointer to an attribute on a specific node. The purpose of plugs is to allow a user query or set a value, create, remove or query a connection. In Maya C++ API there is a special class for this called MPlug.

A data block (see Figure 2.4) is a storage for the data for the attributes and plugs being received or sent by the node. A data handler is a smart pointer into the data stored in the datablock. The validity of a data block lasts only during the compute method.

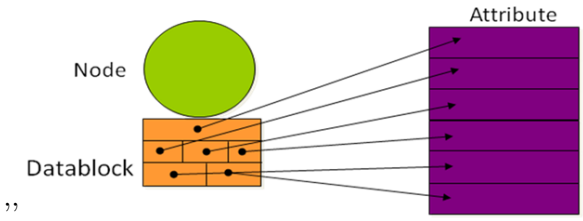


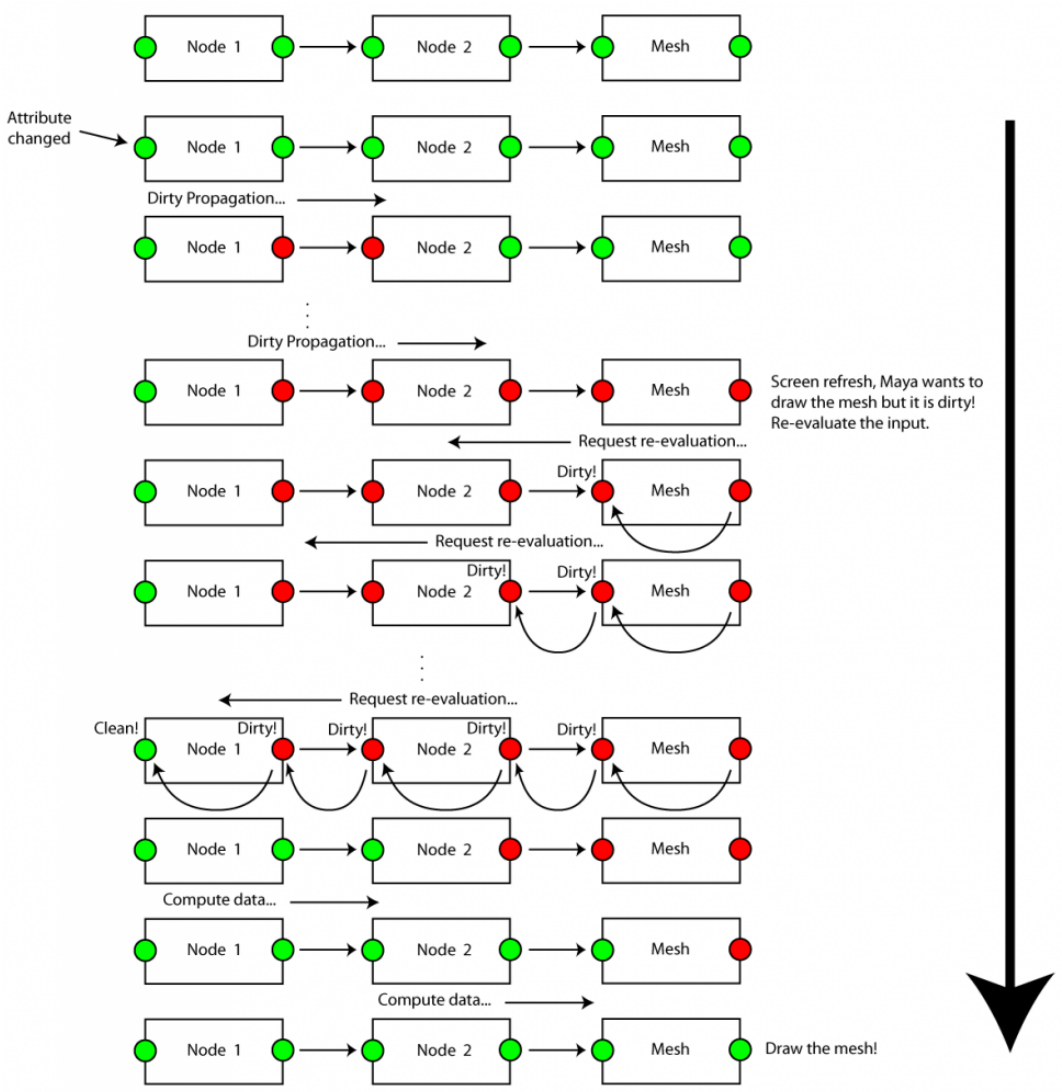
Figure 2.4: A data block.

Nodes also have the compute function which is not displayed on the screen. It takes one or several attributes as inputs and computes one or several outputs.

In case of changing an attribute the DG will calculate the new data only when it needs to do this. This is being done by marking with a *dirty* flag node's inputs and outputs. If an input value is changed, the dependent outputs and the connections to these outputs will be marked *dirty*. This process recurs till the time when the end of the DG is reached. However nothing is recalculated until Maya sends a request for it. Then Maya checks if a node's output is *dirty*. In this case Maya tells the node to get itself evaluated. After processing this the node detects some of its outputs to be *dirty* and asks any connected input node to re-evaluate. The loop will continue until the inputs and outputs are marked as „clean“ (see Figure 2.5). [11] These statuses are stored in a node's data block.

2.1.5 Maya's export plug-ins

Before looking into process of implementing X3D export plug-in we should take a look at some existing Maya's export plug-ins. Here I will make a comparative analysis of OBJ and



,,

Figure 2.5: The change of the node's attribute's flag [11].

VRML export plug-ins. They were chosen for a comparison because of their functionality which is similar to X3D/X3DOM export plug-in implementation.

2.1.5.1 OBJ export plug-in

OBJ files are text-based and support polygonal geometry. In contrast with VRML and X3D/X3DOM OBJ doesn't use tree-hierarchy. Obj files contains information about vertices, texture coordinates, vertex's normals and polygonal faces.[2] All the vertices are specified in lines starting with the letter *v*, texture coordinates - in lines starting with *vt* and the lines starting with *vn* specify the normals. Lines starting with *f* contain information about the each polygonal face. Figure 2.6 illustrates the GUI of the Maya's OBJ export plug-in. Export options can be chosen in the *File Type Specific Options* submenu. The default value for each of the item is switched to *On*.

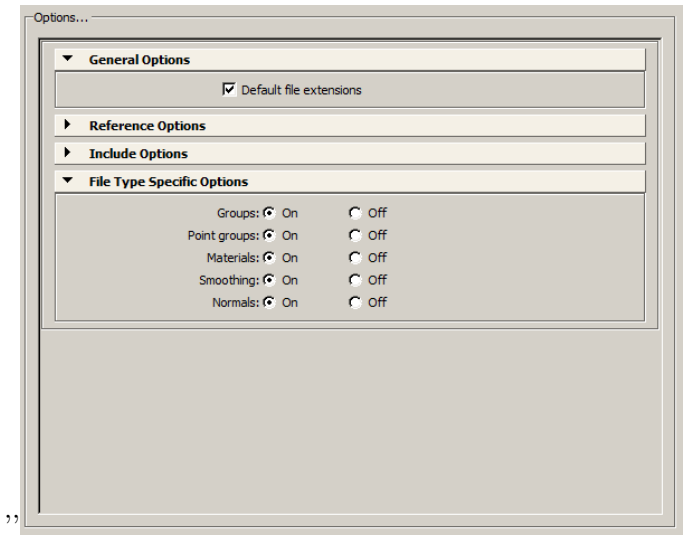


Figure 2.6: The Maya's OBJ export plug-in's GUI.

2.1.5.2 VRML export plug-in

VRML (Virtual Reality Modeling Language) is a standard and file format that is used for interactive representation of 3D vector graphics. [3] Apart from specifying 3D models, materials, textures, sounds and lighting it is also possible to add user interactivity and animations with the use of event handling semantics. The GUI of the Maya's VRML export plug-in is demonstrated on Figure 2.7 - it consists of fours submenus where a user can specify:

- animation options (see Figure 2.8) such as the start and end, the number of frames per second, what do you want to animate, if you want the animation to be looped;
- export options (see Figure 2.9) - for example, define hierarchy, tessellation, if all the objects should be exported or only picked or active ones, if the cameras and lights should be exported as well;

- texture options (see Figure 2.10) - texture size and max size ;
- vrml2 options (see Figure 2.11) - the chosen kind of navigation, its speed, precision, texture path, it also allows to define a path to the script which should be runned after this.

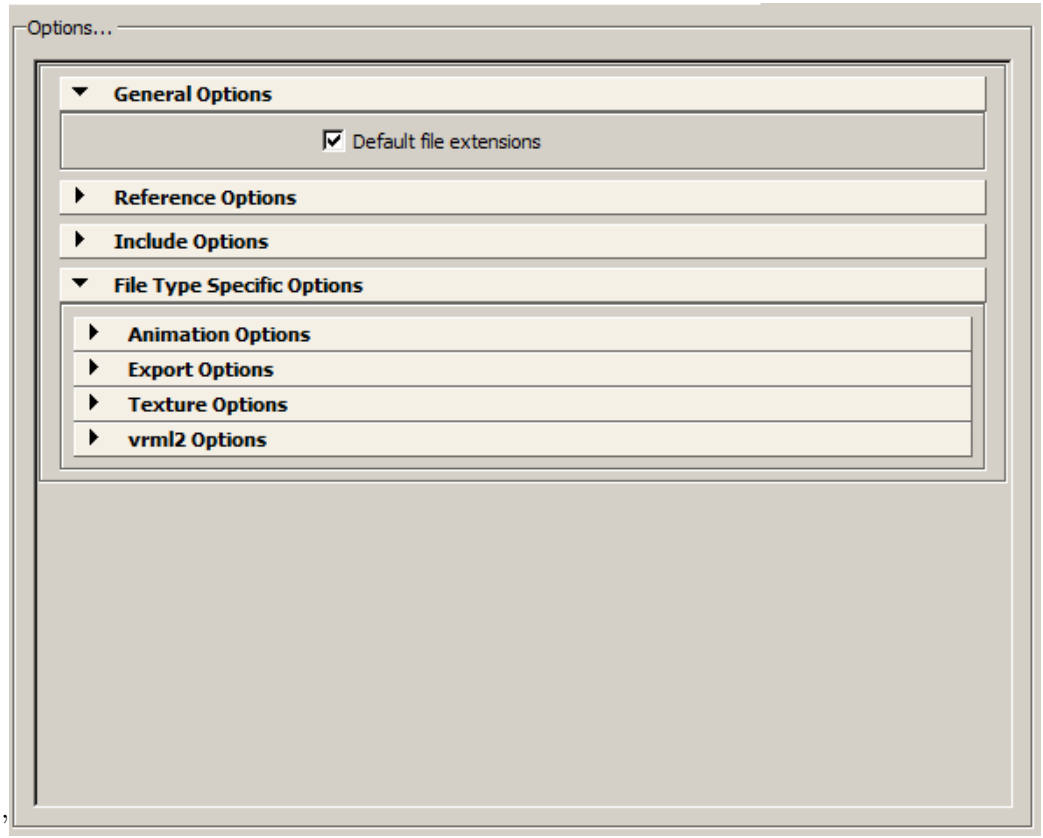


Figure 2.7: The Maya's VRML export plug-in's GUI.

2.2 X3D/X3DOM

X3D (Extensible 3D Graphics) is a royalty-free open standard XML-like file format and run-time architecture for representation of 3D scenes and objects using XML. [5] It is the successor to VRML (the Virtual-reality modeling language) which is the standard for transmitting 3D content across a network environment. [8]

X3DOM is an open source JavaScript framework used for creating 3D scenes and objects in Web pages. [7] It is based on browser technology so there is no need in external plug-ins to display X3DOM scenes in contradistinction to pure X3D – displaying a X3D scene requires a special browser or player such as BS Contact or Instant Reality.

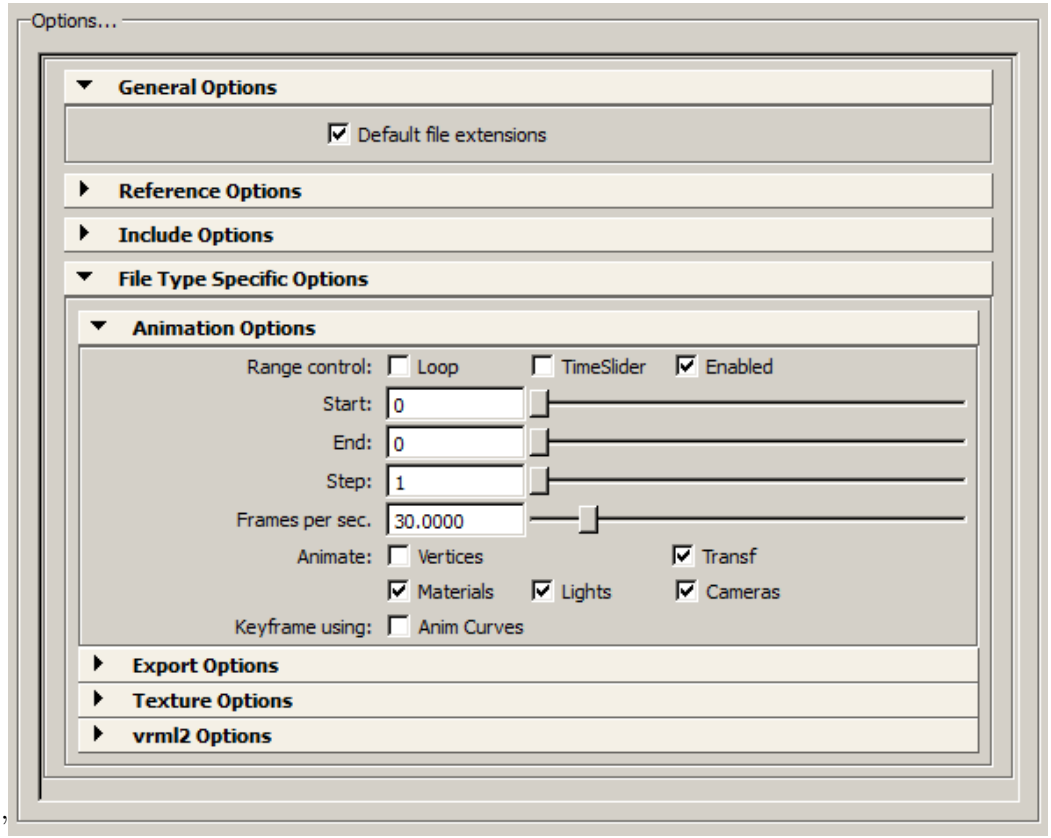


Figure 2.8: The Maya's VRML export plug-in's GUI - Animation options.

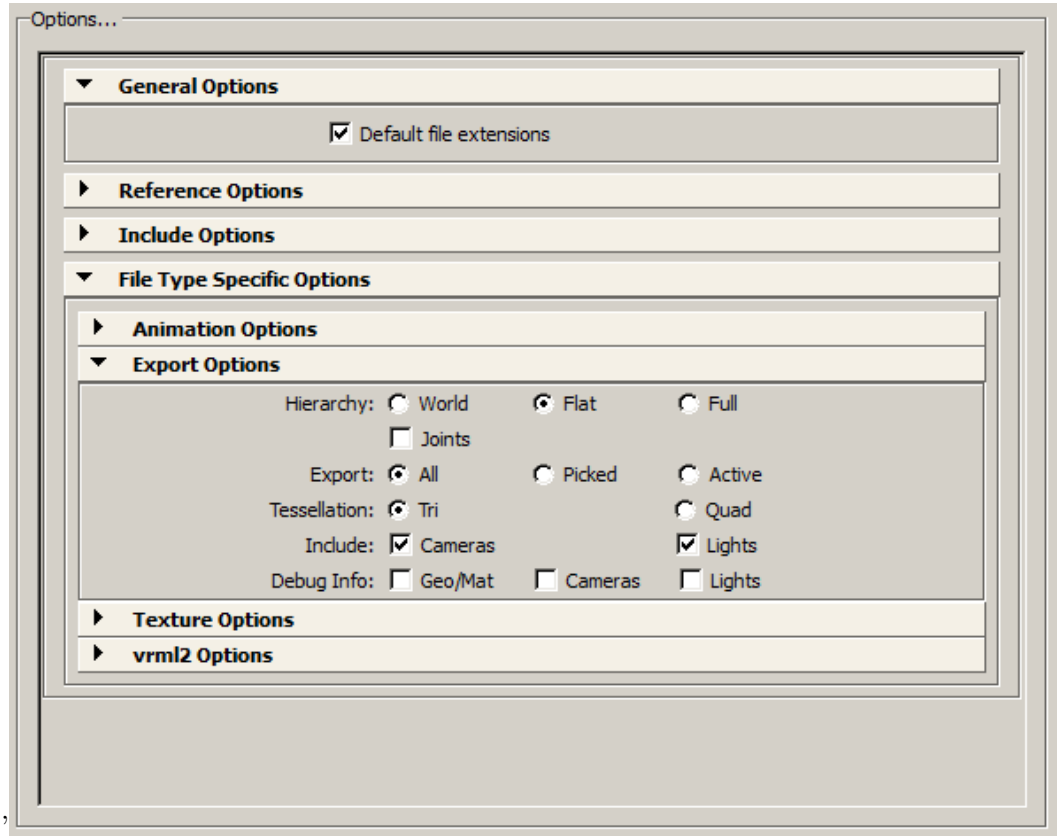


Figure 2.9: The Maya's VRML export plug-in's GUI - Export Options.

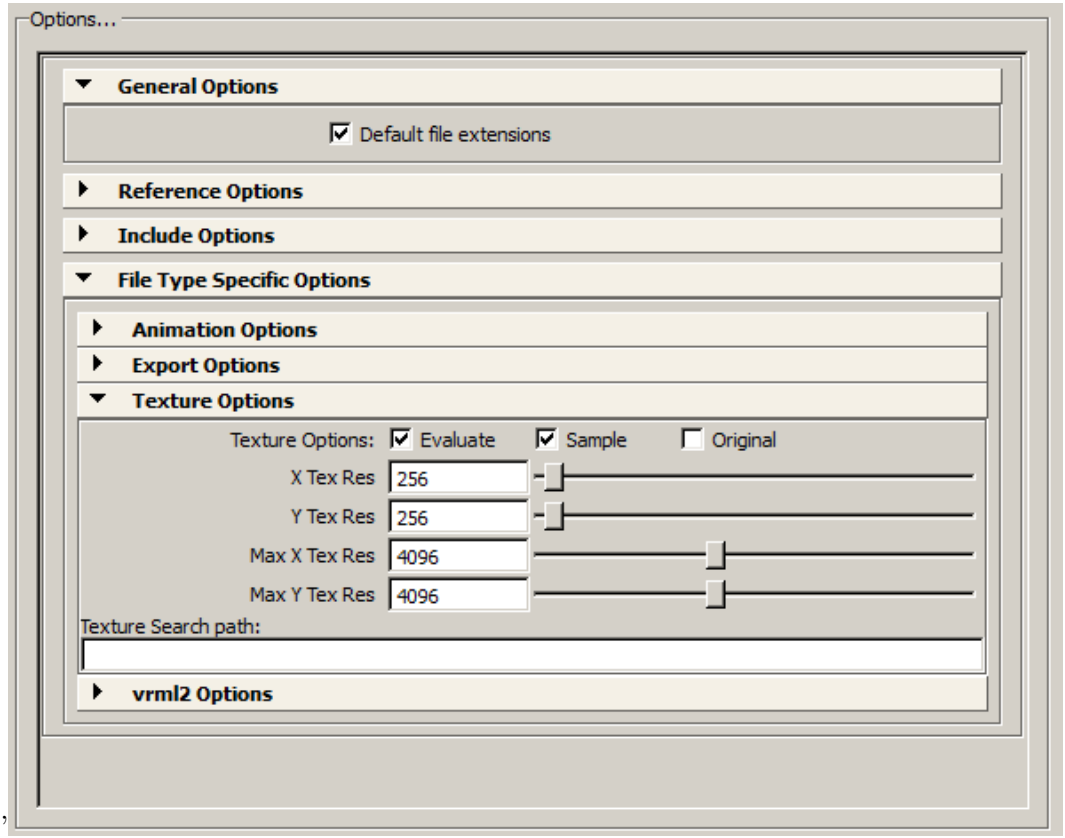


Figure 2.10: The Maya's VRML export plug-in's GUI - Texture options.

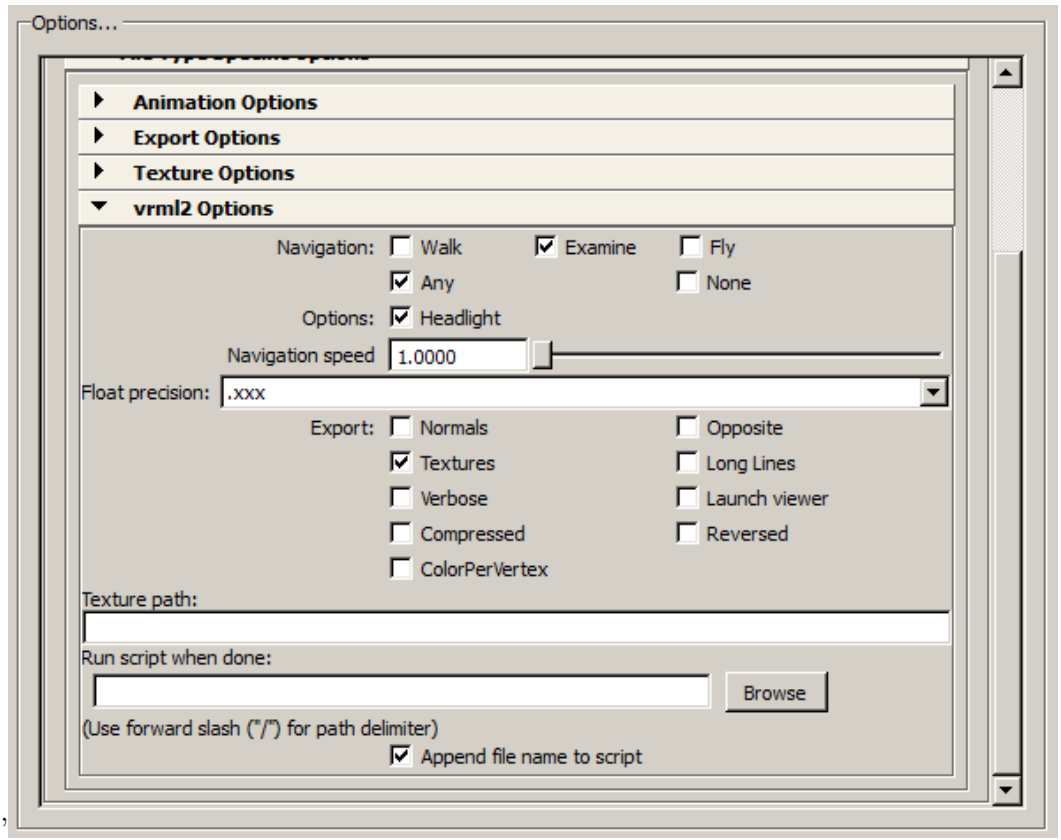


Figure 2.11: The Maya's VRML export plug-in's GUI - vrml2 Options.

Listing 2.1: Example of an X3D scene consisting of a cube

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.0//EN" "http://www.
   web3d.org/specifications/x3d-3.0.dtd">
<X3D version='3.0' profile='Interchange'>
 <Scene DEF='scene'>
  <Group>
   <Viewpoint DEF='persp' description='persp' position='28.0 21.0
      28.0' fieldOfView='54.43'/>
   <Viewpoint DEF='top' description='top' position='0.0 100.1 0.0'
       fieldOfView='54.43'/>
   <Viewpoint DEF='front' description='front' position='0.0 0.0
      100.1' fieldOfView='54.43'/>
   <Viewpoint DEF='side' description='side' position='100.1 0.0
      0.0' fieldOfView='54.43'/>
   <Transform DEF='pCube1' translation ='0.0 0.0 0.0' >
    <Shape>
     <Appearance>
      <Material DEF='lambert1' diffuseColor='0.5 0.5 0.5'
         transparency='0.0' ambientIntensity='0.0'/>
     </Appearance>
     <Box size='4.5 3.9 5.0 '/>
    </Shape>
   </Transform>
  </Group>
 </Scene>
</X3D>
```

Listing 2.2: Example of an X3DOM scene consisting of a cube

```
<html>
 <head>
  <title> A cube </title>
  <script type='text/javascript' src='http://www.x3dom.org/
     download/x3dom.js'>
        </script>
  <link rel='stylesheet' type='text/css' href='http://www.x3dom.
     org/download/x3dom.css'>
        </link>
 </head>
  <body>
   <h1> A cube </h1>
   <x3d width='600px' height='400px' >
    <scene>
     <shape>
      <appearance>
```

```
        <material id="color" diffuseColor='1 0 0'>
        </material>
       </appearance>
       <box></box>
      </shape>
     </scene>
    </x3d>
 </body>
</html>
```

2.3 X3D

2.3.1 X3D Node

X3DNode is main building block of X3D scene and abstract structure that is extended by every X3D component. Main X3D components are:

- Grouping components
- Geometry components
- Appearance component
- Viewpoint component
- Light components
- Sensors
- Routes

2.3.2 Grouping and a hierarchical scene

Grouping is widely used for easier manipulation over collection of nodes. In X3D there are several grouping nodes:

- Anchor
- Billboard
- Collision
- Group
- LOD
- Switch
- Transform

The scene hierarchy is a hierarchy consisting of nodes that are laid out in the tree-like structure representing parent-child connections between the nodes. X3D scene structure is very similar to Maya scene structure. Figure 2.12 illustrates both of them: the pyramid and the pipe are grouped while the pyramid is the parent of several objects.

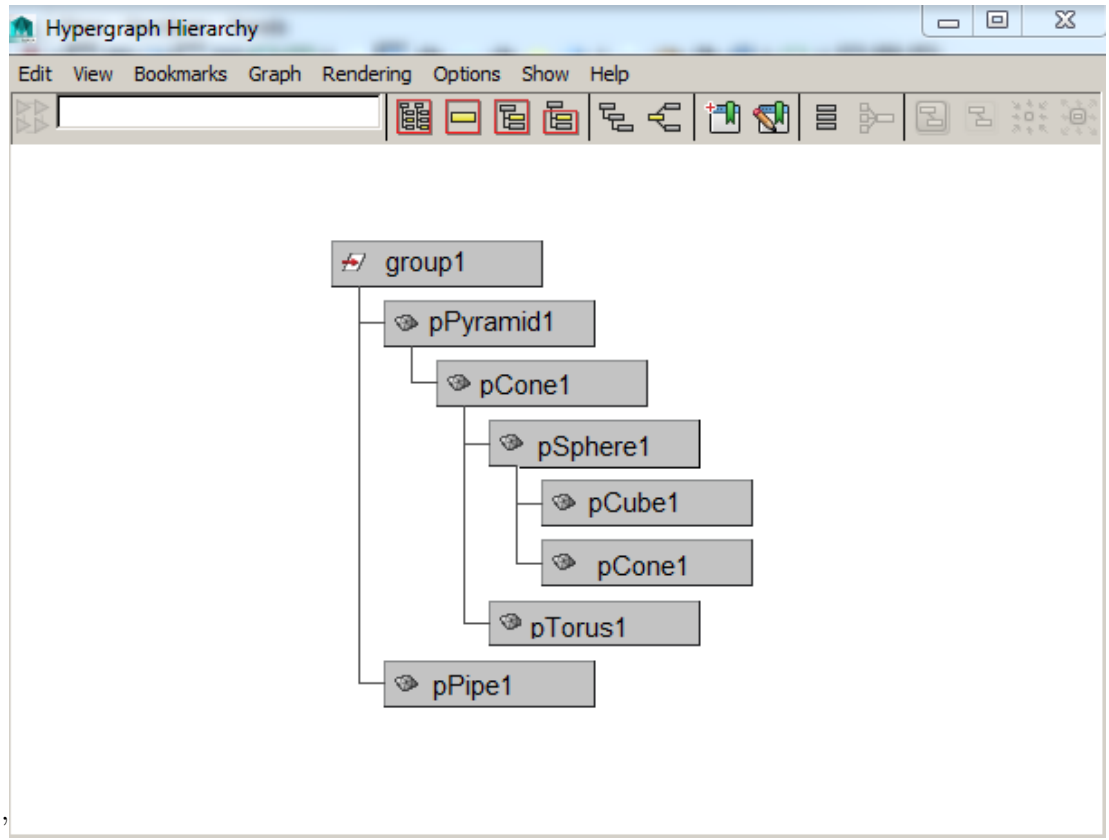


Figure 2.12: The scene hierarchy shown in Maya's Hypergraph editor.

By making node a child of another node, user establishes the *Parent-Child Hierarchy*. Then when a user, for example, moves the parent object, the child moves with its parent so its position towards the parent object doesn't change. In X3D this relationship can be represented by putting the child node into the *Transform* node of the parent.

Grouping is used for creating a parent transformation node for collection of objects as well as for graphically arranging nodes in scene. For instance: there is a scene consisting of separate models that together represent a car. Each model is a part of the car: a carcass, wheels, headlights, interior. *Grouping* in this particular example allows a user to apply transformations (translation, scale, rotation) for collection of models as if it was a single object.

2.3.3 Geometry components

In X3D all the 3D models are defined with *Shape* nodes. *Shape* node should be included under *Transform* node which is used for translation, rotation and scale. Additionally all shape nodes have to be placed under grouping node called *Scene*. (see Listing 2.3).

The Shape node always consists of a geometry node and can also include an Appearance node.

Listing 2.3: A *Shape* node example

```
<Scene DEF='scene'>
 <Group>
  <Transform>
   <Shape>
    <Appearance>
     <Material DEF='lambert1' diffuseColor='0.5 0.5 0.5'/>
    </Appearance>
    <Box/>
   </Shape>
  </Transform>
 </Group>
</Scene>
```

The geometry node can be represented by a specific node corresponding to primitive object geometry (Cube,Cylinder e.t.c) or by the *<IndexedFaceSet>* tag. All these shapes have one common feature – SFBool [] *solid* field that returns FALSE if the object should two-sided rendering. This field refers to Maya's shape attribute *Backface Culling* (see Figure 2.13). Turning it 'full' or 'off' sets *solid* to TRUE or FALSE respectively.

For *Box*tag is used for displaying a cube (see Listing 2.4). This node belongs to the self-closing nodes and includes the three-dimensional vector for the object's width, height and depth. Corresponding values for those fields can be found in Maya's Attribute Editor (see Figure 2.14).

Listing 2.4: A *Box* node example

```
<Shape>
    <Box size='5 2 3'/>
</Shape>
```

Spherical shaped objects are represented by the *Sphere* tag. It is a self-closing tag too and also contains the information about the radius of the sphere. The Figure 2.15 illustrates the *Sphere* node.

The geometrical figures of the conical shape are displayed by the self-closing *Cone* tag with the predetermined bottom radius of the cone's base and the height from the base to the apex. Their default values are set to 1.0 and 2.0 respectively. The Figure 2.16 illustrates the *Cone* node.

The *Cylinder* node (see Figure 2.17) specifies a cylinder. The tag includes the radius and the height fields. Besides the node has three SFBool fields that determine if the cylinder's side, top and bottom parts exist. In this case these fields' values are set to TRUE.

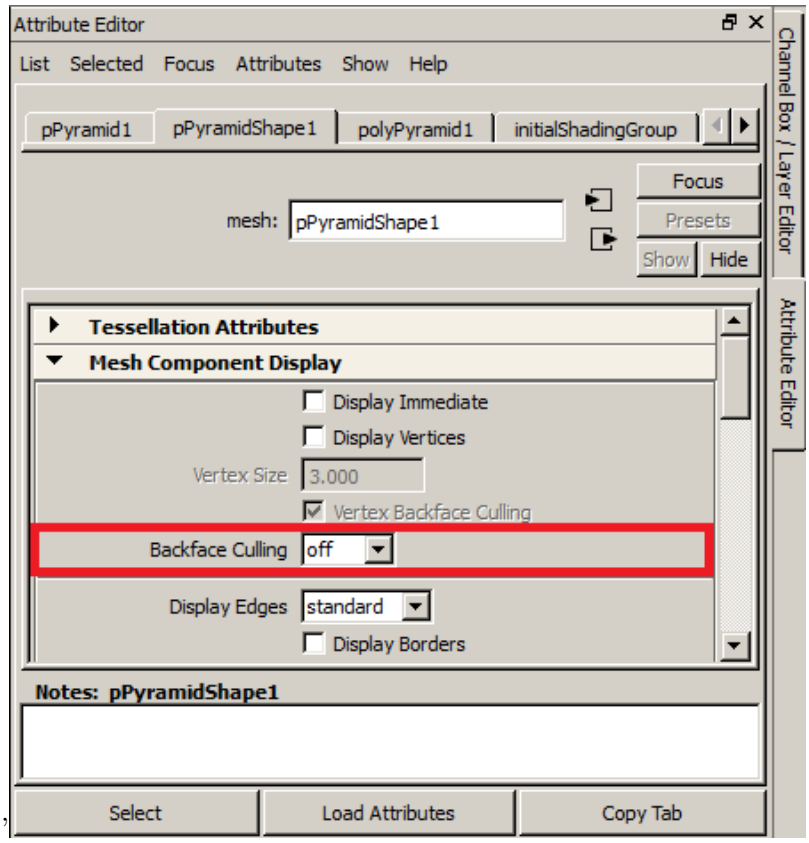


Figure 2.13: Maya's Backface Culling illustration.

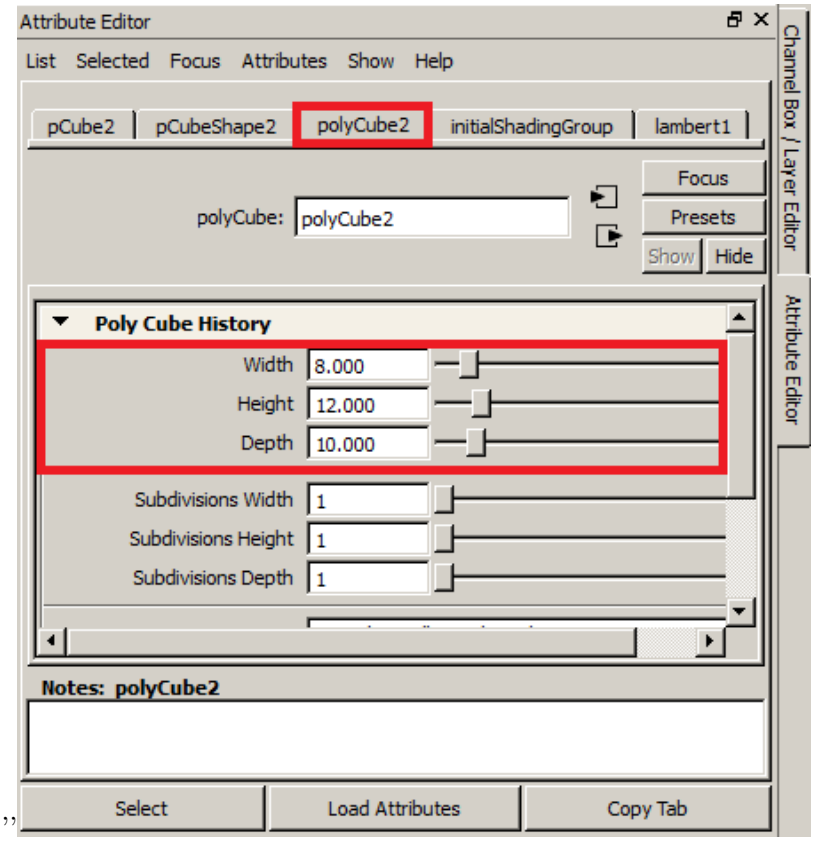


Figure 2.14: Values for the parallelepiped's size.

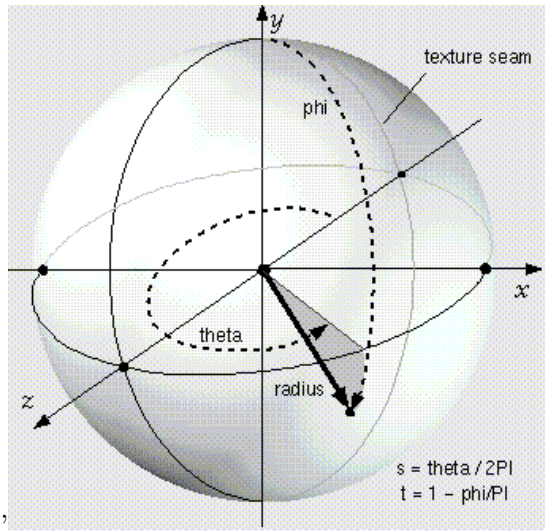


Figure 2.15: The *Sphere* node [5].

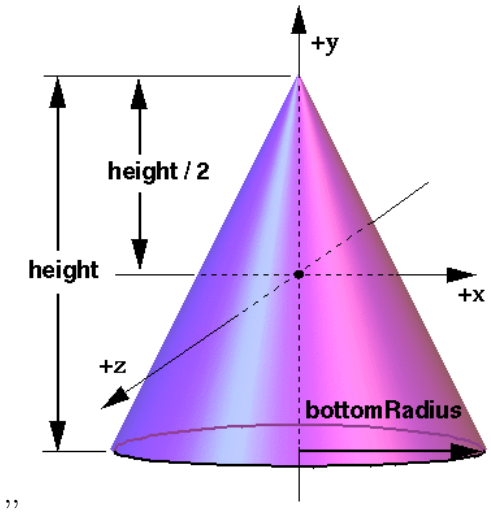


Figure 2.16: The *Cone* node [5].

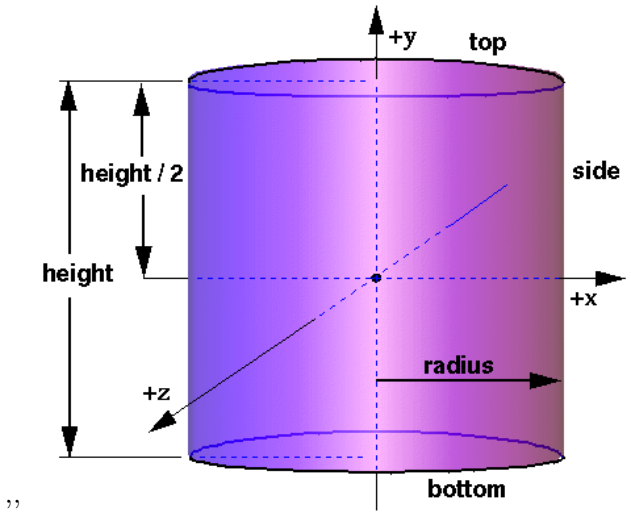


Figure 2.17: The *Cylinder* node [5].

There is another node describing the shape of an object - *IndexedFaceSet*. There are two cases when it is being used. One of them is when a user is creating an object with a complex shape that cannot be described by the basic shapes' nodes. The second one is when the basic shape was changed after the object had been created, e.g. one vertex was moved. The purpose of this tag is to construct faces using the vertices from the *Coordinate* node in a specific order defined in the *coordIndex* field. As can be seen in Listing 2.5, the *coordIndex* field as well as the *texCoordIndex* field always consists of the indices and "-1". This "-1" is used to show the end of the face.

The *TextureCoordinate* node specifies the coordinates of the texture applied on the object's surface.

Listing 2.5: A *IndexedFaceSet* node example

```
<IndexedFaceSet DEF='pCubeShape1' texCoordIndex='0 1 3 2 -1 2 3 5
    4 -1 4 5 7 6 -1 6 7 9 8 -1 1 10 11 3 -1 12 0 2 13 -1'
 coordIndex='0 1 3 2 0 -1 2 3 5 4 0 -1 4 5 7 6 0 -1 6 7 1 0 0 -1 1
    7 5 3 0 -1 6 0 2 4 0 -1'>
 <Coordinate DEF='pCubeShape1Points' point='-2.500 -2.500 2.500
    2.500 -2.500 2.500 -2.500 2.500 2.500 2.500 2.500 -2.500
    2.500 -2.500 2.500 2.500 -2.500 -2.500 -2.500 -2.500 2.500
    -2.500 -2.500'/>
 <TextureCoordinate point='0.375 0.0000.625 0.0000.375 0.2500.625
    0.2500.375 0.5000.625 0.5000.375 0.7500.625 0.7500.375
    1.0000.625 1.0000.875 0.0000.875 0.2500.125 0.0000.125 0.250'/
    >
</IndexedFaceSet>
```

2.3.4 Appearance

The visual properties of the object such as material (see Listing 2.6) or texture (see Listing 2.7) are described by the *Appearance* node located inside the *Shape* tag.

Listing 2.6: A *Material* node example

```
<Appearance>
 <Material DEF='lambert1' diffuseColor='0.5 0.5 0.5' transparency=
    '0.7' ambientIntensity='0.3'/>
</Appearance>
```

Listing 2.7: A *Texture* node example

```
<Appearance>
 <ImageTexture DEF='file1' url='"C:/sun.PNG"'/>
 <Material diffuseColor='0.5 0.5 0.5' specularColor='0.4 0.4 0.4'/
    >
</Appearance>
```

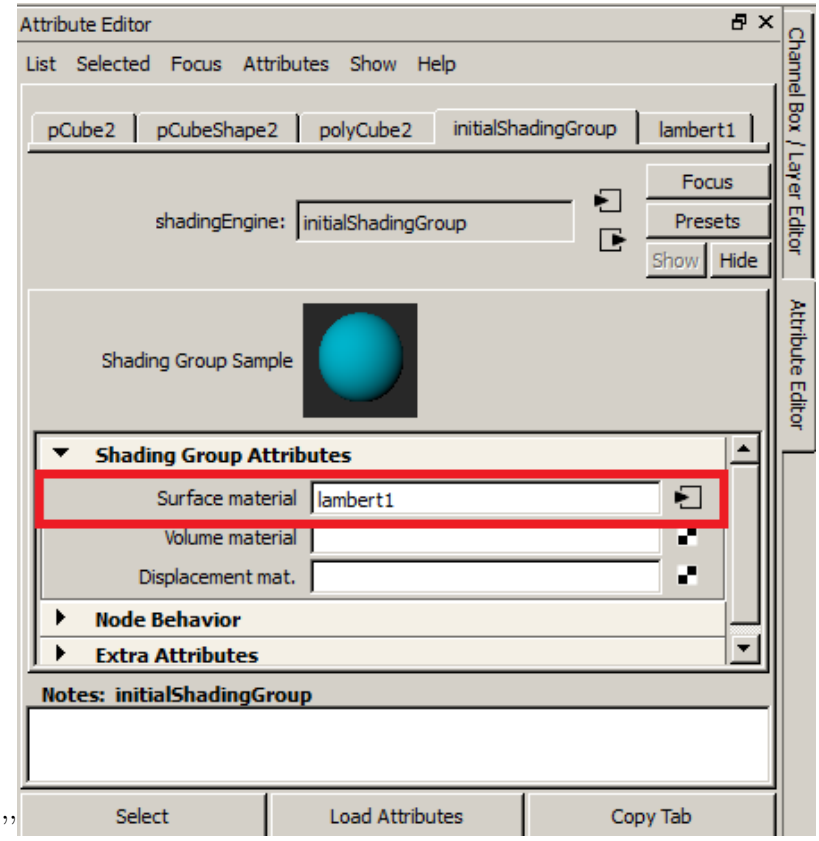


Figure 2.18: Material in Maya.

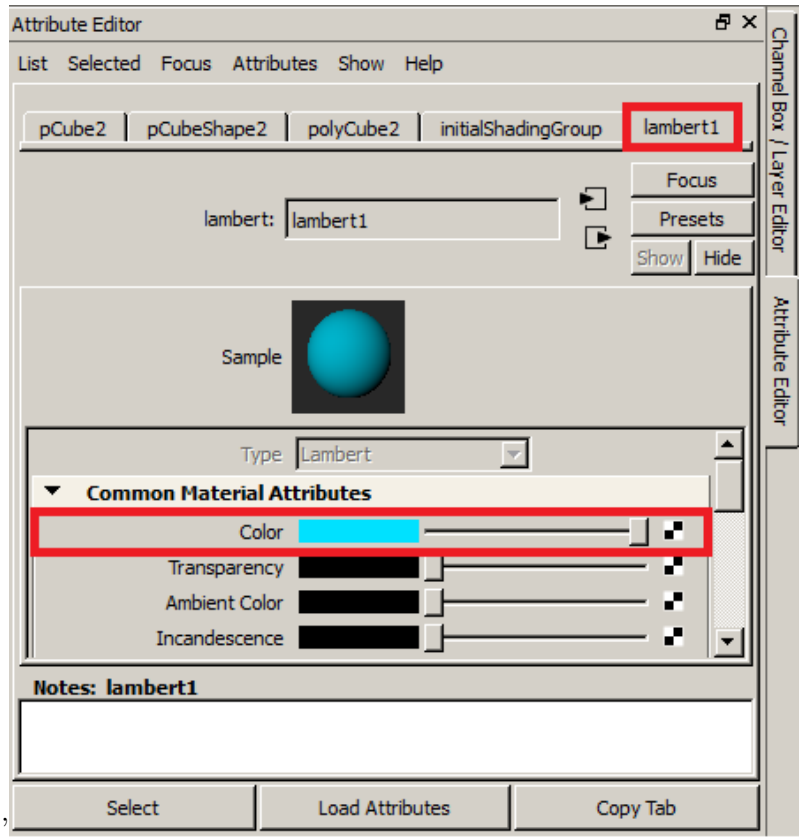


Figure 2.19: Material's attributes 1.

Name of the material is taken from the field *Surface material*. The field *diffuseColor* of the *Material* node takes in the values from the Maya's *Color* field inside the *material* (in this example - *lambert1*) tab (see Figure 2.19).

In order to make the object's appearance correspond to the reality as much as possible, the *Material* node should contain the following attributes: *ambientIntensity*, *transparency*.The values for these fields are taken from the Maya's *material* (in this example - *lambert1*) tab (see Figure 2.20).

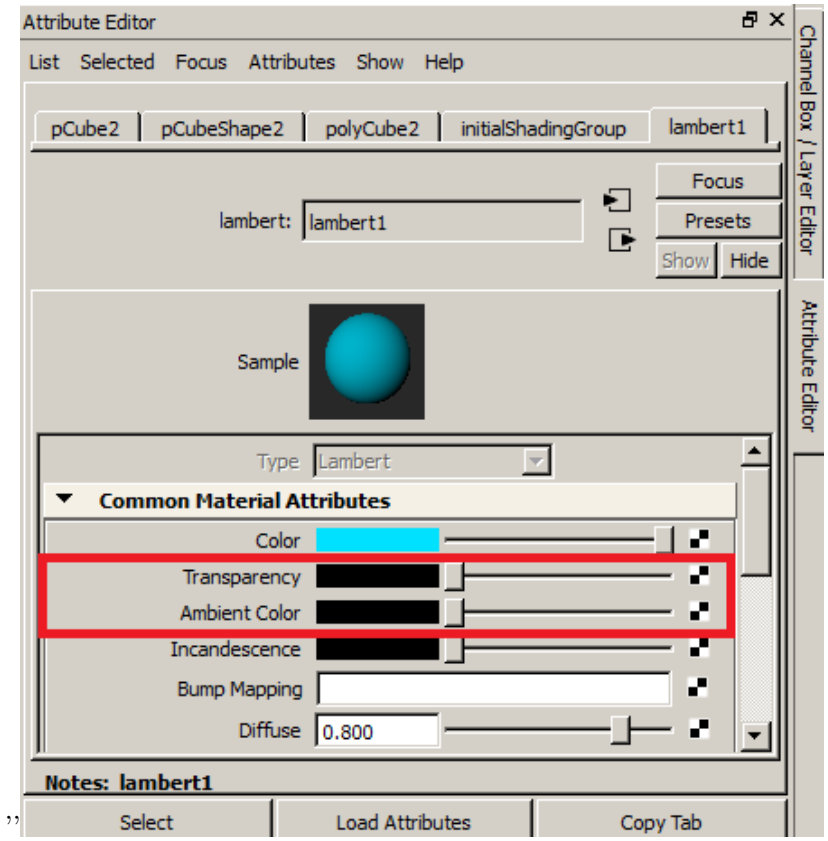


Figure 2.20: Material's attributes 2.

In the second case when a texture is applied to the object's surface the data needed for specifying the *ImageTexture* node are taken from the Maya's *file* (in this example – *file1*) tab (see Figure 2.21). Figure 2.22 illustrates how this tab can be reached.

2.3.5 Viewpoint

Maya's cameras corresponds to the *Viewpoint* nodes. These nodes used to define users view on a scene. Important attributes of the node are the *description*, the *fieldOfView*, *orientation* and the *position*. The *orientation* attribute describes viewpoint's eye vector. The *position* attribute describes viewpoint's location. (see Figure 2.23). Listing 2.8 demonstrates the default cameras and the specified *camera1*.

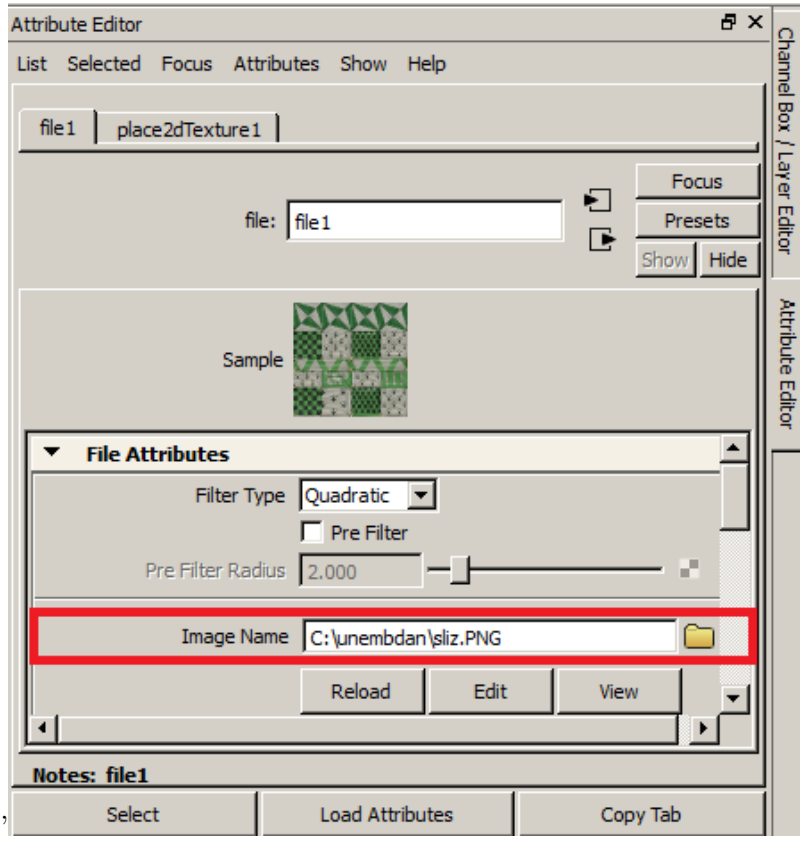


Figure 2.21: Texture file's data

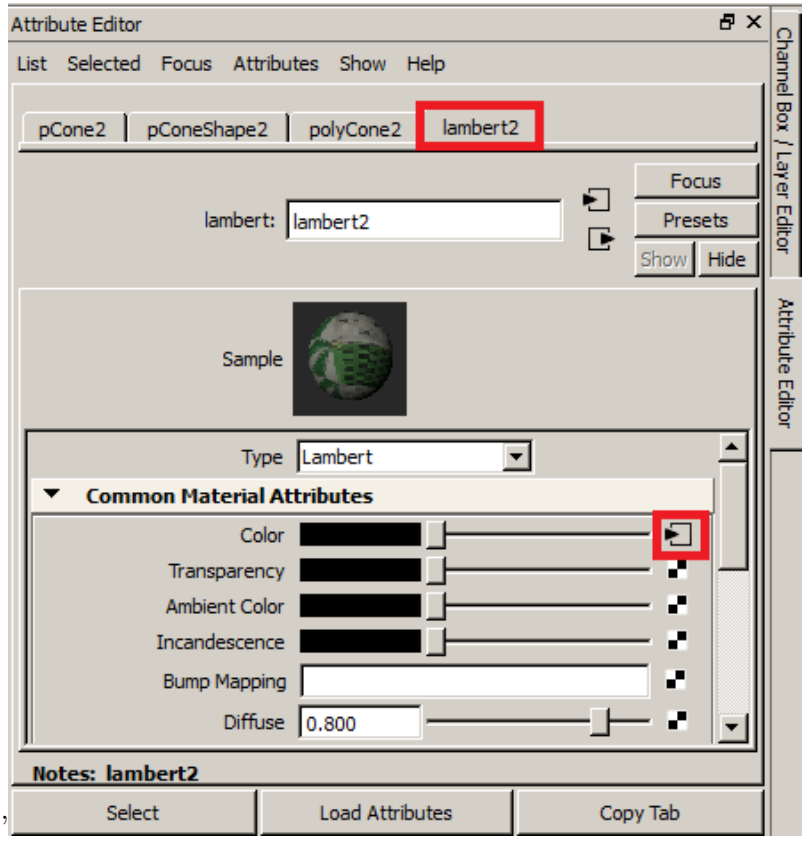


Figure 2.22: Material's attributes 2.

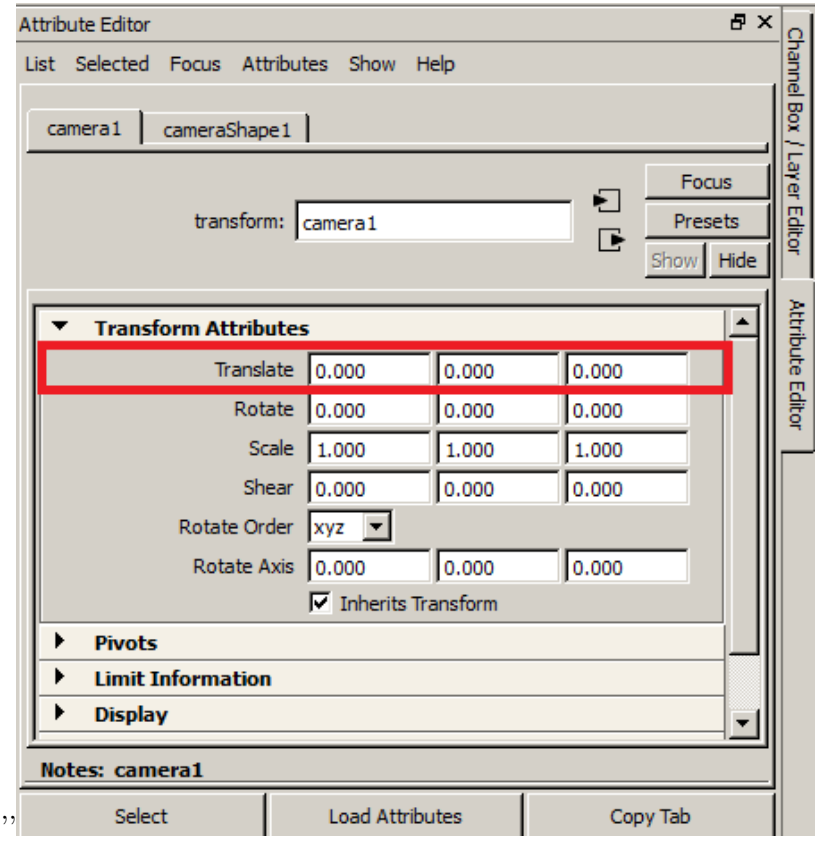


Figure 2.23: Camera information.

Listing 2.8: A *Viewpoint* node example

```
<Viewpoint DEF='persp' description='persp' position='28.000 21.000
    28.000' fieldOfView='54.43'/>
<Viewpoint DEF='top' description='top' position='0.000 100.100
    0.000' fieldOfView='54.43'/>
<Viewpoint DEF='front' description='front' position='0.000 0.000
    100.100' fieldOfView='54.43'/>
<Viewpoint DEF='side' description='side' position='100.100 0.000
    0.000' fieldOfView='54.43'/>
<Viewpoint DEF='camera1' description='camera1' position='0.000
    8.847 12.133' fieldOfView='54.43'/>
```

2.3.6 Light

X3D Light components represent light sources.

The *DirectionalLight* node represents the parallel rays of light going in the same direction (see Figure 2.24).

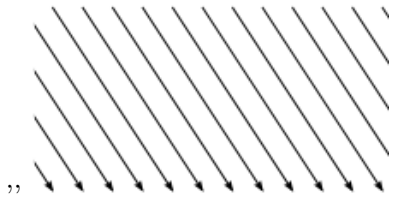


Figure 2.24: Directional light illustration [6].

The node's important fields are *color*, *direction* and *intensity*. The values for the *color* and *intensity* fields can be accessed from the *light's shape* in Maya (see Figure 2.25). The *direction* of the light corresponds to the rotation of the light in Maya's light's *transform* tab (in this example – *directionalLight1* tab). This is demonstrated on the Figure 2.26.

The *PointLight* node represents the light source of the point shape. As can be seen in Figure 2.27, the rays go in all directions.

The *color* , the *radius* and the *intensity* fields take in data from the corresponding fields of Maya's *light's shape* tab. The values for the *location* field can be found in the Maya's light's *transform* tab.

The *SpotLight* node consists of several fields (see Figure 2.28) including the *beamWidth*, *color*, *cutOffAngle*, *direction*, *intensity*, *location* and *radius* fields. The values for them can be found either in Maya's Attribute Editor or can be calculated with the help of the other fields.

2.3.7 Animation

In X3D object's changes in animation scope are described by three components :

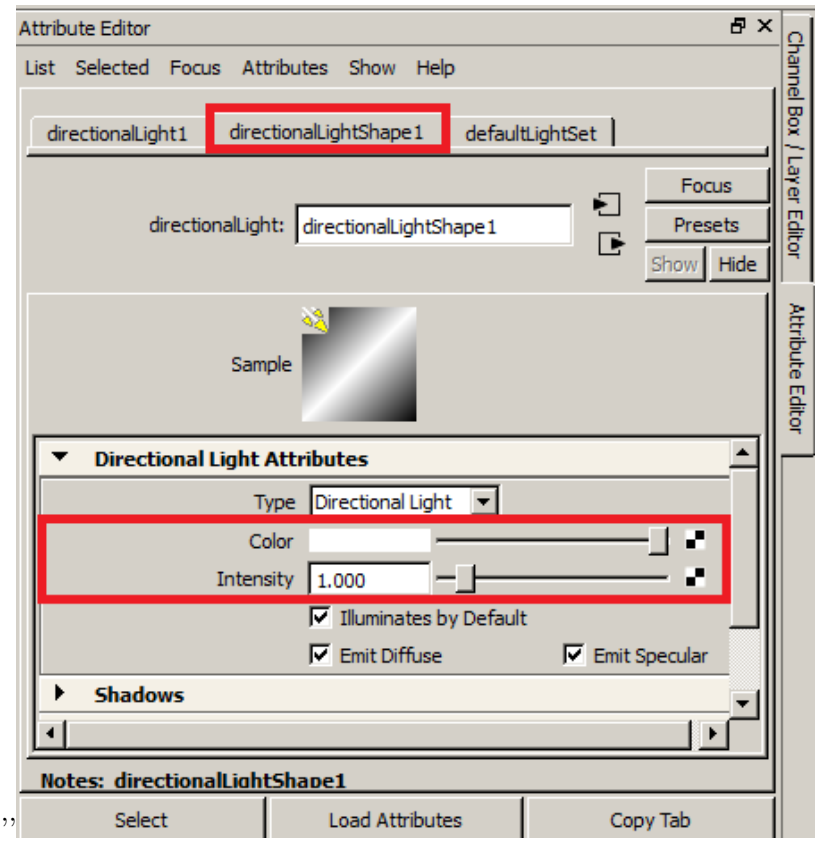


Figure 2.25: Directional light in Maya 1.

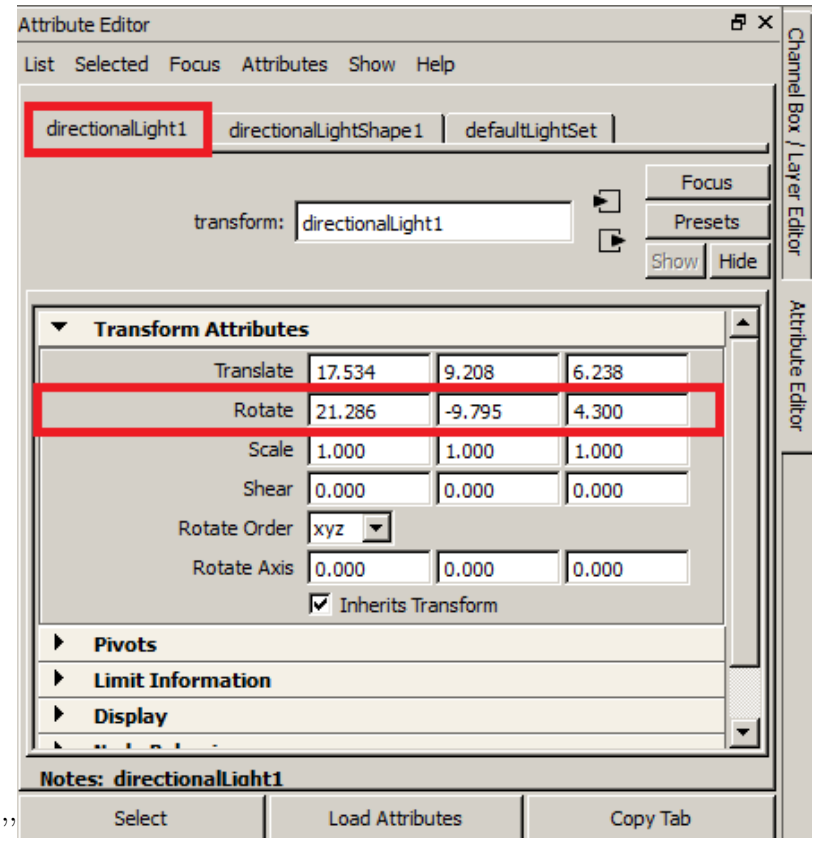
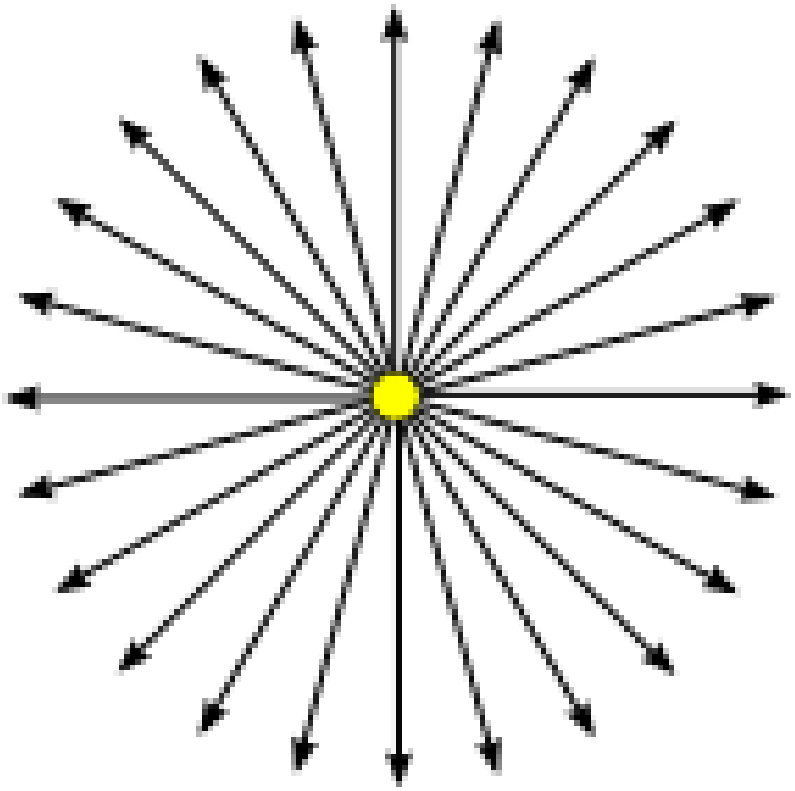
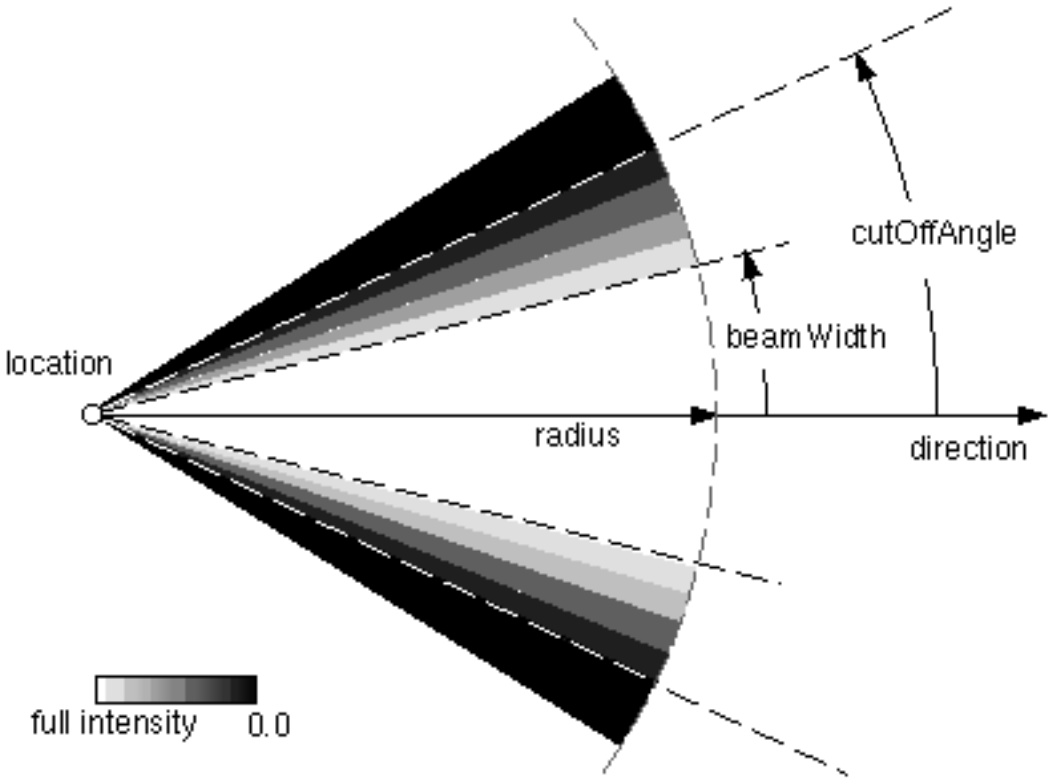


Figure 2.26: Directional light in Maya 2.



,,

Figure 2.27: Point light illustration [6].



,,

Figure 2.28: The *SpotLight* node [5].

- Sensors
- Interpolators
- Routes

TimeSensor node is used as a timer for animation in X3D. It has such attributes as *cycleInterval* and *loop* fields. *CycleInterval* defines length of animation in milliseconds. Attribute *loop* indicates whenever animation should run infinitely or only once. *TimeSensor* also generates events as time passes, those events then used for iteration over *Interpolators*' values

OrientationInterpolator and *PositionInterpolator* nodes are holding values of object's changes according their position on a timeline. The first one is being used when a user wants to show the rotation of the object. The second node represent scaling and translation.

The interpolator's attributes are *key*, *keyValue*. The field *key* corresponds to the Maya's *animation frames* (see Figure 2.29) and the *keyValue* field takes in an array of rotation, scale or translation values of the object in the current frame (see Figure 2.30).

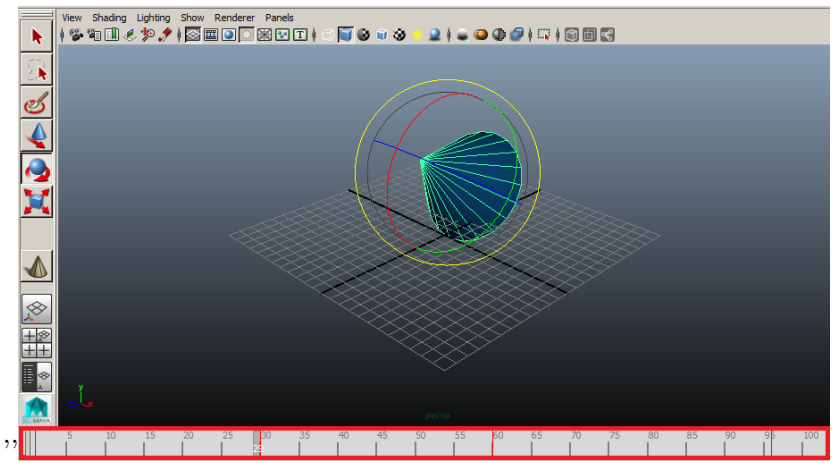


Figure 2.29: The animation frames.

ROUTE components (illustrated in Listing 2.9) provide connection between inputs and outputs of the *TimeSensor*, *Interpolator* and *Transform* components.

Listing 2.9: Animation of scaling in X3D example

```
<Group>
 <Transform DEF='BallTransform'>
  <Shape>
   <Sphere/>
        </Shape>
 </Transform>
 <TimeSensor DEF='CLOCK' cycleInterval='2.0' loop='true'/>
 <PositionInterpolator DEF='BALLSIZE' key='0.0 0.2 0.65 1.0'
    keyValue='1.0 1.0 1.0 1.5 1.5 1.5 1.1 1.1 1.1 1.0 1.0 1.0'/>
```

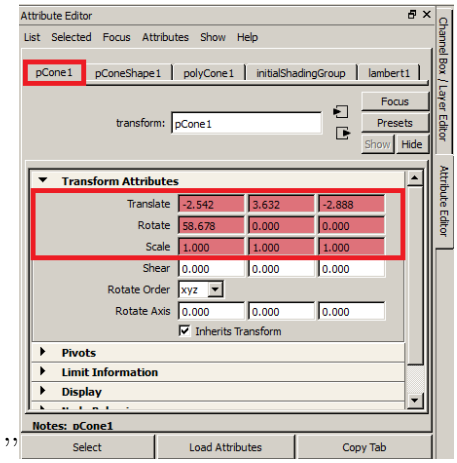


Figure 2.30: The values for the *keyValue* field shown in Maya.

```
 </Group>
<ROUTE fromField='value_changed' fromNode='BALLSIZE' toField='
    set_scale' toNode='BallTransform'/>
<ROUTE fromField='fraction_changed' fromNode='CLOCK' toField='
    set_fraction' toNode='BALLSIZE'/>
```

2.4 Mapping Maya nodes to X3D nodes

Despite of the high variety of Maya nodes almost all of them can be described in X3D notation. Most of the Maya and X3D nodes differ in attributes but semantically serve the same purpose. The following table 2.1 demonstrates how an element represented in Maya can be mapped to X3D, where the last column named "+/-" means whether the node or the attribute can be accessed directly in Maya to be mapped to X3D or not.

Category	Element	Maya	X3D	+/-
Shape [2.3.3]	**Cube**	polyCube	Box	+
Shape [2.3.3]	**Sphere**	polySphere	Sphere	+
Shape [2.3.3]	**Cone**	polyCone	Cone	+
Shape [2.3.3]	**Cylinder**	polyCylinder	Cylinder	+
Attribute	Size	height/width/depth	size	+
Attribute	Radius	Radius	radius	+
Attribute	Height	Height	height	+
Attribute	Bottom radius	Radius	bottomRadius	+
Shape [2.3.3]	**Other shapes**	Another shape node	IndexedFaceSet	+
Attribute	Texture Coord indices	UV index	texCoordIndex	+
Attribute	Coordinate indices	vertex index	coordIndex	+
Attribute	Coordinates	vertex position	Coordinate point	+
Attribute	Texture coordinates	UV coordinate	TextureCoordinate	+
Attribute	Crease angle	Crease angle	creaseAngle	-
Appearance [2.3.4]	**Material**	initialShadingGroup	Material	+
Attribute	Definition	Surface material	DEF	+
Attribute	Color	Color	diffuseColor	+
Attribute	Transparency	Transparency	transparency	+
Attribute	Ambient color	Ambient color	ambientIntensity	+
Camera [2.3.5]	**Camera**	Camera	Viewpoint	+
Attribute	Definition	tranform	DEF	+
Attribute	Description	tranform	description	+
Attribute	Position	Translate	position	+
Attribute	Orientation	Rotate	orientation	+
Attribute	Field of view	Angle of view	fieldOfVie	+
Lighting [2.3.6]	**Directional light**	Directional light	DirectionalLight	+
Attribute	Difenition	transform node	DEF	+
Attribute	Color	Color	color	+
Attribute	Direction	Rotate	direction	+
Attribute	Intensity	Intensity	intensity	+
Lighting	**Point light**	Point light	PointLight	+
Lighting	**Spot light**	Spot light	SpotLight	+
Attribute	Cone Angle	Cone angle	beamWidth	+
Attribute	Location	Translate	location	+
Attribute	CutOff angle	-	cutOffAngle [4.6]	+
Attribute	radius	Cone angle	radius	+
Animation [2.3.7]	Animation	Animation curve	Interpolator	+
Attribute	Key	Key frame	key	+
Attribute	Value	Translate/Rotate/Scale	keyValue	+

Table 2.1: Mapping Maya nodes to X3D nodes

Chapter 3

Design

This chapter is an output of analysis made on existing Maya plug-ins both their GUIs and source code.

3.1 Plug-in's software architecture

Main .cpp file will consist of four classes (see Figure 3.1)

- Main exporter class that will communicate with Maya inner classes. This class will be run as soon as export option in Maya will be chosen.
- Utility class that will contain methods for working with Maya and X3D data.
- Wrapper class for whole X3D scene.
- Wrapper class for X3D node.

3.2 Graphical user interface

For the purpose of allowing a user to define the precision of the values that will be written into X3D/X3DOM nodes' fields as well as the definition of the output file's format there should be a GUI. This GUI (see Figure 3.2) will contain *radio button* allowing to choose output file format and *slider* for defining a precision. (see 3.2).

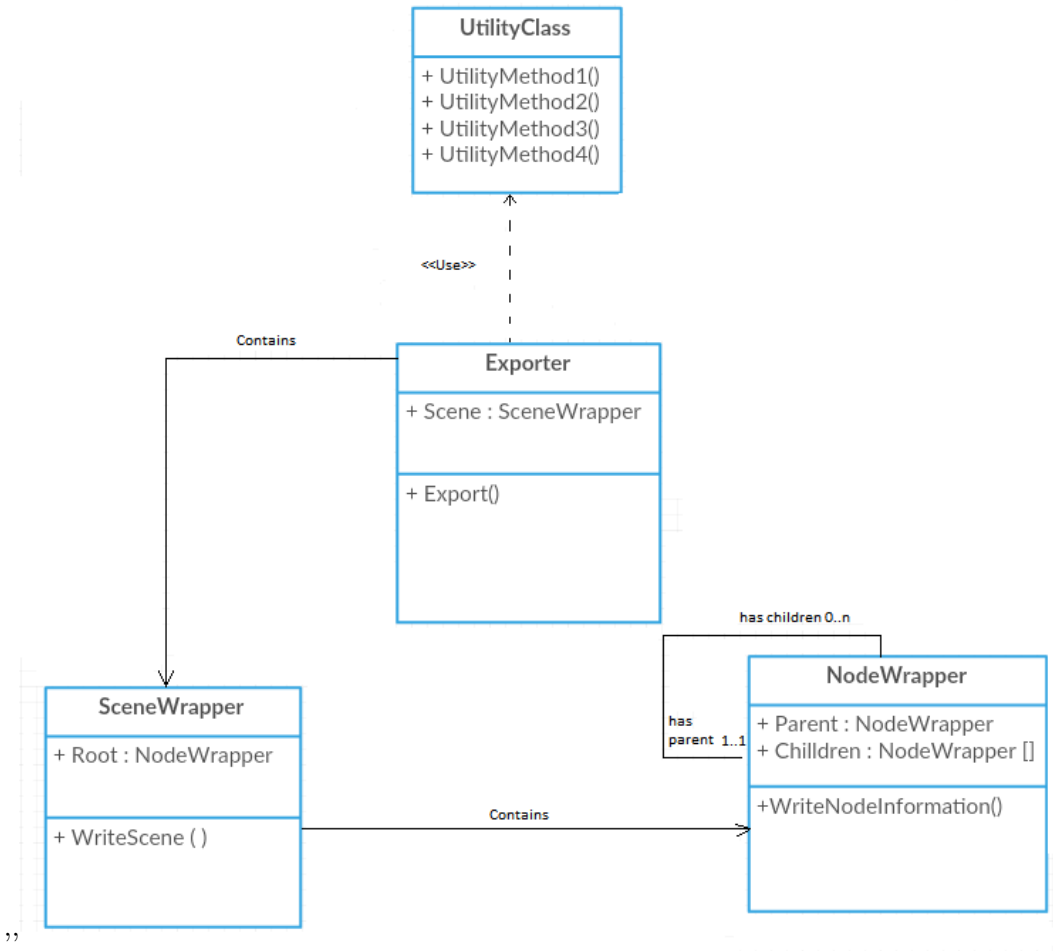


Figure 3.1: Plug-in class diagram

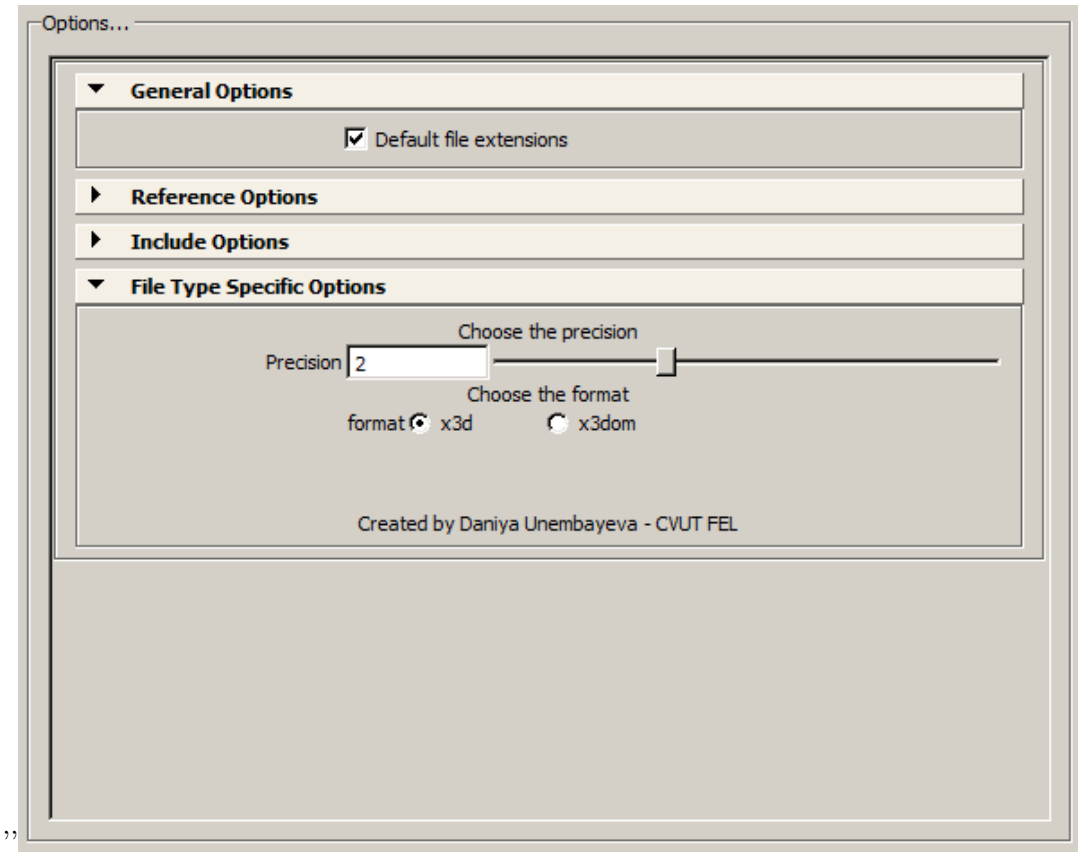


Figure 3.2: The GUI of the plug-in.

Chapter 4

Implementation

4.1 General information

The C++ Maya API export project consists of the main executive file ("exporter2015Cmd.cpp"), necessary .dll and .h files, a MEL script ("exportOptions.mel"). Output of this project is a builded .mll file that is then used as an export plug-in in Maya.

4.1.1 *exporter2015Cmd.cpp*

This source file's logic is implemented with four C++ classes:

- x3dExporter
- x3dUtil
- SceneStructure
- SceneStructureNode

4.1.1.1 *x3dExporter* class

This class derives Maya API's standard *MPxFileTranslator* class which allows to implement an export plug-in for Maya. The main job of *MPxFileTranslator* is done in two of its methods: *writer* and *reader*. Every time Maya is importing or exporting objects either *reader* or *writer* is called respectively. We won't mention *reader* features as this work describes building an export plug-in. The *writer* method contains logics behind decision whenever *exportSelected* or *exportAll* method is called. [1]

4.1.2 *exportOptions.mel*

The script handles two actions - *post* and *query*. The user interface is being created in the first action, in the second the string of options builded and then passed to the *writer* method of the *x3dExporter* class. [4]

4.1.3 *SceneStructureNode* class

As the Maya's scene and X3D/X3DOM structures differ from each other we need to have a wrapping class (*SceneStructureNode* class) which represents each object of the scene for convenience of the exporter's work. Every *SceneStructureNode* object has a pointer to a *MObject* it represents, attributes *parent, children* and *type* corresponding X3D notation standard and methods for working with *MObject*.

4.1.4 *SceneStructure* class

This class is a wrapper for the whole scene. It also keeps the pointer to the *SceneStructureNode* root element and helpful methods.

4.1.5 *x3dUtil* class

It is a class with utillity methods that are helping to obtain information from Maya and correctly pass it to X3D format.

4.2 Querying attributes

The method *findPlug* of the *MPlug class* is being used for getting the information about a Maya node's attributes. Listing 4.1 illustrates an example of getting a light's color.

Listing 4.1: Querying the color attribute using a plug

```
MPlug colorPlug = lightDependNode.findPlug("color", true, &stat);
double r = colorPlug.child(0).asDouble();
double g = colorPlug.child(1).asDouble();
double b = colorPlug.child(2).asDouble();
```

4.3 Choosing between primitive geometry tag and IndexedFaceSet

According bachelor's thesis assignment cube-shaped, sphere-shaped, cylinder-shaped and cone-shaped objects should be exported using the specific basic geometry node. We will take a look at this on the example of exporting a cube. First of all, we are checking if there is a node *polyCube* (see Figure 4.1) that generates this primitive (see Listing 4.2).

Listing 4.2: Determines if the node has the polyCube generator

```
boxShape.hasFn(MFn::kPolyCube);
```

Secondly, if the object was changed and no longer corresponds to the original object generated by the generator node, plug-in has to figure out if it is the original object or the modified (decide if the geometry will be printed to the output file using the *Box* tag or the *IndexedFaceSet* tag).

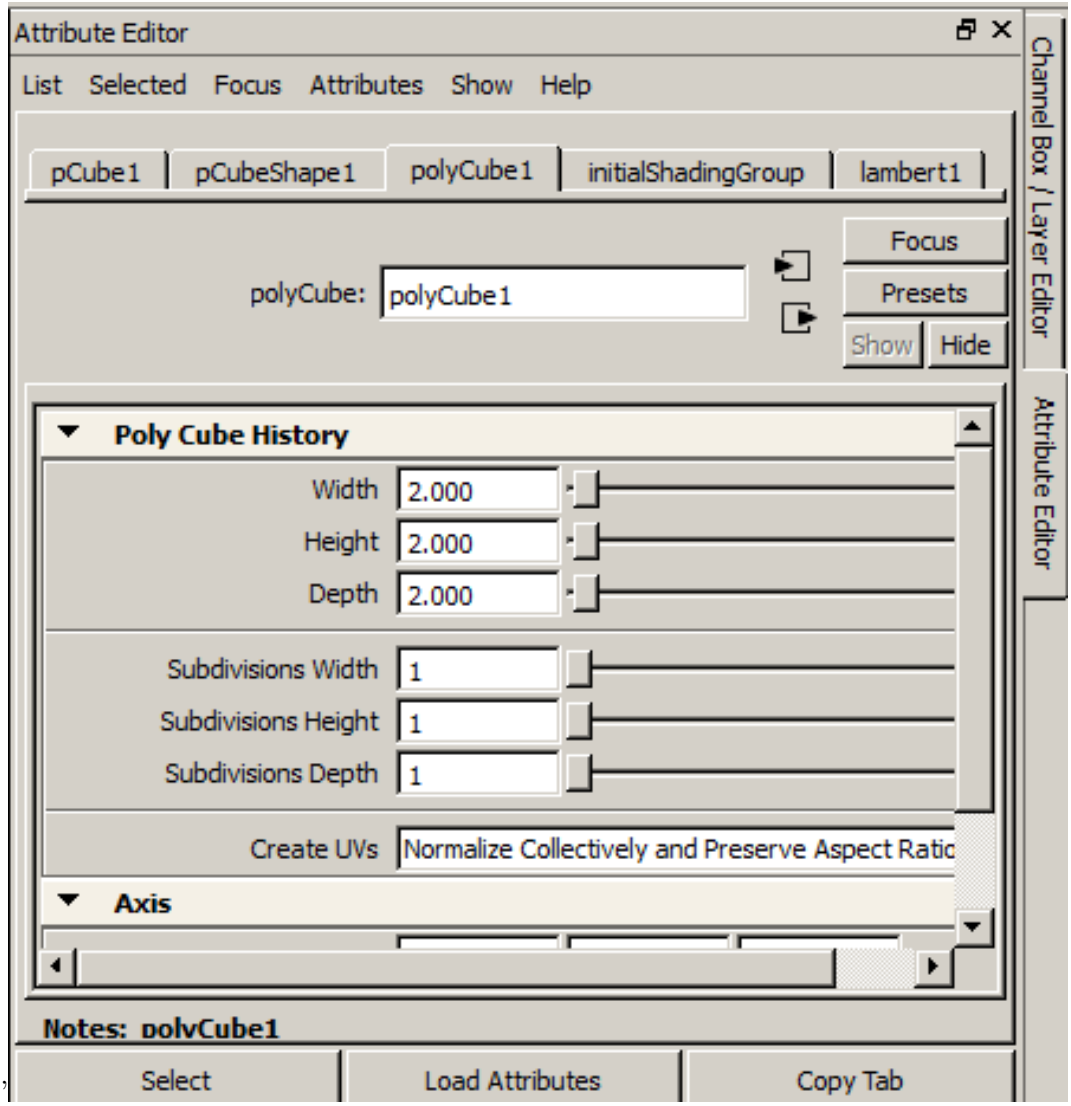


Figure 4.1: The *polyCube* generator in Maya.

Thirdly, because of the fact that metadata about the modifications of the object might not correspond to the reality there was used a radical method in which we take the data of original object (vertices and UVs) and compare them with the same type of data of the object reconstruct by the parameters of the *polyCube* (see Listing 4.3).

Listing 4.3: Reconstructing the object using the generator node's parameters such as height, width, depth

```
MGlobal::executeCommand("CreatePolygonCube;setToolTo␣
    CreatePolyCubeCtx;polyCube␣␣-n␣_test_shape␣-h␣2␣-w␣2␣-d␣2");
```

The comparison is held in *areMeshesEqual* method of the *x3dUtil* class - the data of both objects are being written in two separate strings and then compared.

In case the objects are equal we use the *Box* tag, otherwise - the *IndexedFaceSet* tag.

4.4 Precision

According to the bachelor's thesis assignment the plug-in should allow to set global precision of all numeric data that are being exported to X3D/X3DOM. In spite of this requirement there are some cases in which choosing precision of zero digits after the decimal point will lead to incorrect user experience and errors. A perfect example will be exporting animation.

Exported animation keys (see Listing 4.4 with zero digits after the decimal point will be written in the output file as demonstrated in Listing 4.5, which will completely break the animation.

Listing 4.4: Original keys

```
key='0␣0.25␣1'
```

Listing 4.5: Original keys

```
key='0␣0␣1'
```

4.5 Animation

In order to obtain the necessary information about the animation in the scene we need to get the animation curve node. Firstly, all the animated plugs on the active selection should be found and then collected with the *findAnimationPlugs* method. Use of MEL command is shown in the Listing 4.6, where xxxx is the name of the object, yyyy is the operation (translation, rotation, scale) and A stands for the axis transform values (X, Y ,Z ,W), the plug-in will obtain all neccesary information for defining the changes in the whole scope of the animation curve.

Listing 4.6: Getting information for the *keyValue* field

```
keyframe -q -vc xxxx_yyyyA
```

To obtain appropriate key values the plug-in takes every keyFrame's time and normalizes it according to the maximal length of the time line (see Listing 4.7).

Listing 4.7: Normalizing key value.

```
float currentKeyFrameTime = animCurve.time(
    currentKeyFrameNumber - 1).value(); //getting time of
    the current keyFrame
float normalizedKeyFrame = currentKeyFrameTime/
    total_maximal_time; // normalizing value
```

4.6 Lights

Below is described a process of getting *Light* atrributes from Maya and modifying them for X3D. In case of directional and spot light there is a need of get the direction vector. For this purpose I use rotation matrices. As the default order in Maya is *xyz* firstly I multiply the RotationX matrix $\begin{pmatrix} 1 & 0 & 1 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$ by the default X3D SpotLight/DirectionalLight direction vector $\begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$ and get a result vector $\begin{pmatrix} r1 \\ r2 \\ r3 \end{pmatrix}$. The next RotationY matrix $\begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix}$ is being multiplied by the $\begin{pmatrix} r1 \\ r2 \\ r3 \end{pmatrix}$ vector. The new result vector is $\begin{pmatrix} r4 \\ r5 \\ r6 \end{pmatrix}$. The last RotationZ matrix $\begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$ is being multiplied by the $\begin{pmatrix} r4 \\ r5 \\ r6 \end{pmatrix}$. The final result vector is the needed direction vector. To get the angles used in the matrices I use a special MEL command (see Listing 4.8), where *name* is the current item's name (e.g. *pCone1*), cosX is $\cos\alpha$, sinX is $\sin\alpha$, etc.

Listing 4.8: Getting the values for the angles

```
MDoubleArray result;
MString command = "xform -q -r -ro ";
command+= name;
MGlobal::executeCommand(command, result);
/*we need to convert degrees to radians and get their sine, cosine
    values.*/
double cosX = cos(result[0]*PI/180);
double sinX = sin(result[0]*PI/180);
double cosY = cos(result[1]*PI/180);
double sinY = sin(result[1]*PI/180);
double cosZ = cos(result[2]*PI/180);
double sinZ = sin(result[2]*PI/180);
```

In case of a spot light there are two fields, values for which are not accessible using *findPlug* method, - *beamWidth* and *cutOffAngle*. Listing 4.9 illustrates the values of *beamWidth* *cutOffAngle* depending on the *penumbra angle's* value. Penumbra is a softer and lighter part

of a cast shadow, which forms due to a fraction of light getting past the object which is casting said shadow.

Listing 4.9: Getting values for the *beamWidth* and *cutOffAngle* fields

```
MPlug coneAnglePlug = f.findPlug("coneAngle",true,&stat);
MPlug penumbraAnglePlug = f.findPlug("penumbraAngle",true,&stat);
double coneAngle = coneAnglePlug.asDouble();
double penumbraAngle = penumbraAnglePlug.asDouble();
double beamWidth;
double cutOffAngle;
if (penumbraAngle >= 0) {
 beamWidth = coneAngle;
 cutOffAngle = coneAngle + penumbraAngle;
} else {
 double beamWidth = coneAngle - penumbraAngle;
 double cutOffAngle = coneAngle;
}
```

Chapter 5

Testing

The first part of this chapter is a rough showcase of the plug-in's capabilities that are delivered in a form of stability tests. The second part is dedicated to the usability testing. For opening X3D files we use InstantPlayer, X3DOM (locally) - Mozilla Firefox as Google Chrome and Opera do not allow to view textures on the meshes due to their security policy that refuses every image/file that is not server by the server.

5.1 Stability tests

5.1.1 Shapes

To test the export of the basic shapes we make several tests:

1. **Basic and complex shapes** to make sure all basic-shaped objects are being exported using their corresponding nodes and the remaining are written to the file with the help of the *IndexedFaceSet* node. For this test we create a cube, a sphere, a cylinder, a cone, a pyramid, a torus and a pipe in Maya (see Figure 5.1) and export the whole scene to X3D and then to X3DOM with precision set to "3". As a result (see Figure 5.2)) we get correctly exported shapes.

2. **A modified basic shape** with a translated vertex is expected to be printed into the output file as an *IndexedFaceSet* node. For this we are modifying the cone's top vertex (illustrated on Figure 5.3), then export the selected object to X3D and after that to X3DOM with precision set to "3". Figure 5.4 demonstrates that the exported modified object was exported correctly.

The tests' result is that the plug-in correctly exports basic shapes, modified basic shapes, complex shapes to X3D and X3DOM both.

5.1.2 Appearance

To make sure the export of the appearance of an object works properly we will export a complex-shaped textured object, a textured primitive and an object with the *transparency*

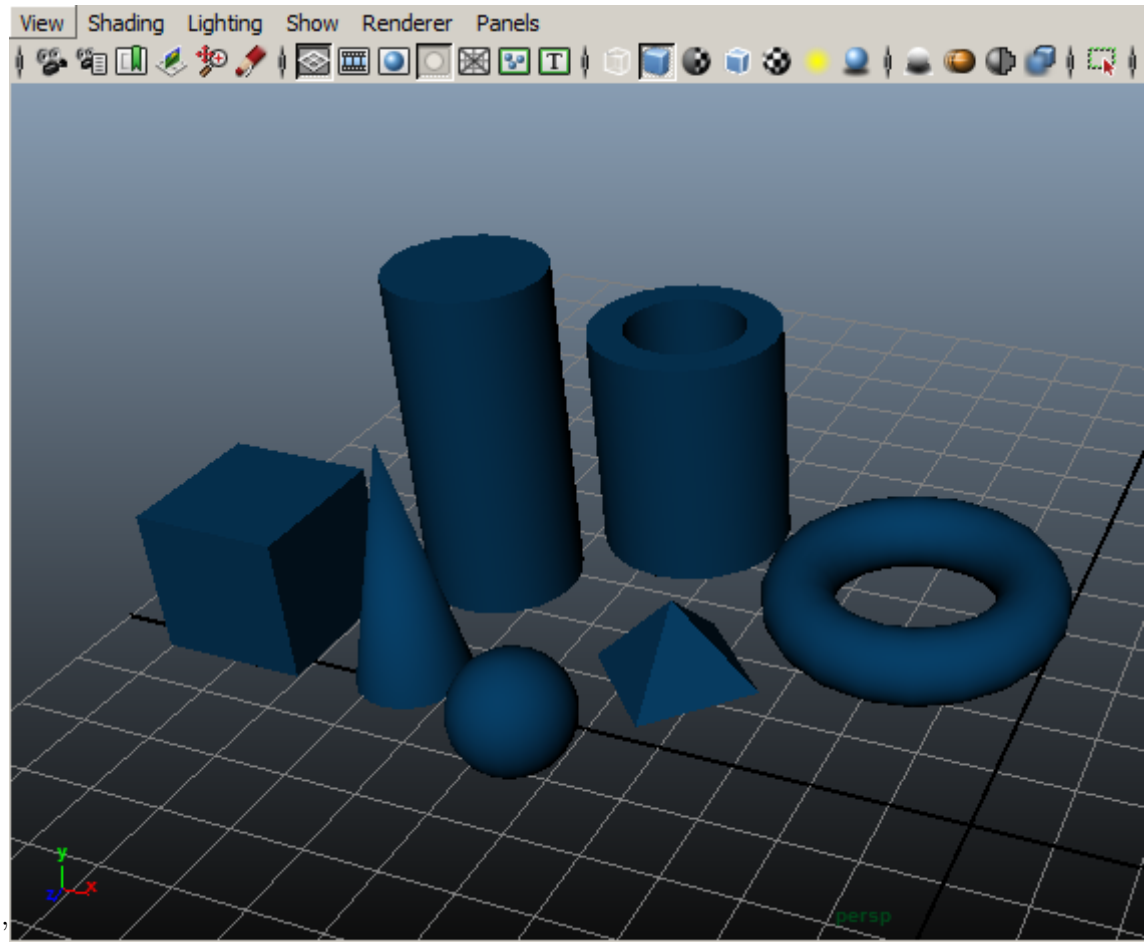


Figure 5.1: The basic and complex shapes to be exported in Maya.

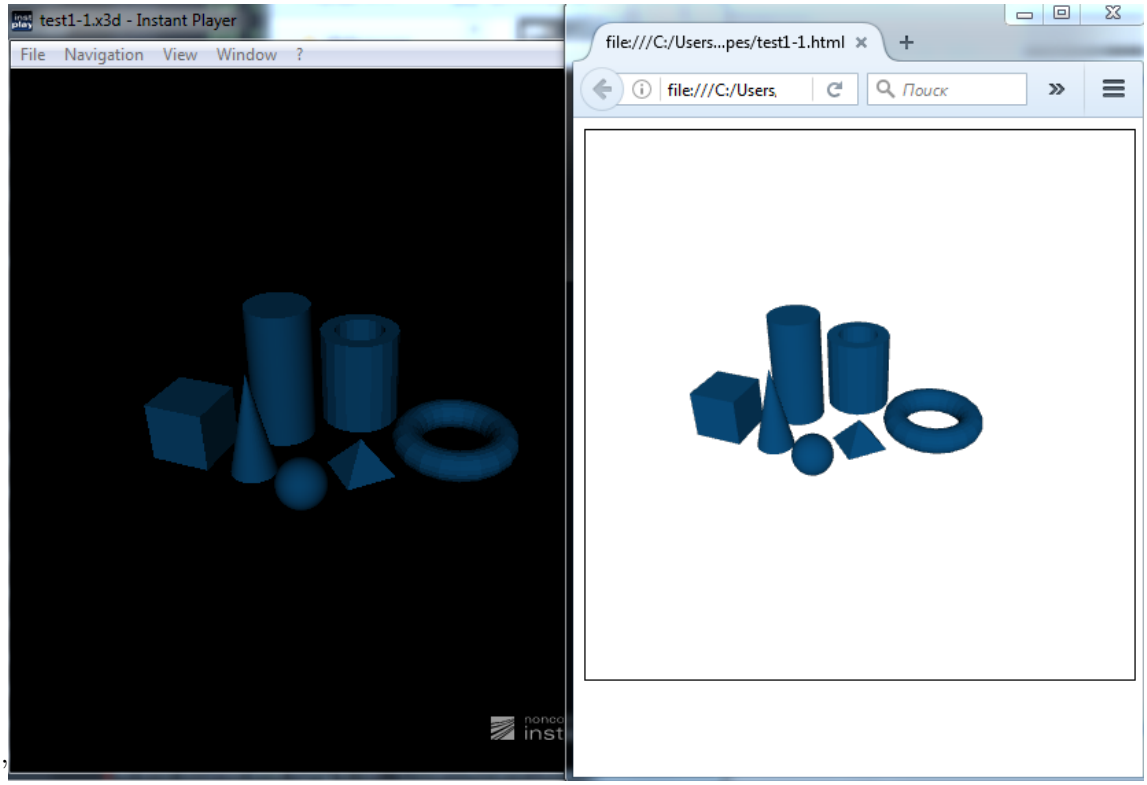


Figure 5.2: The basic and complex shapes' export results in X3D and X3DOM.

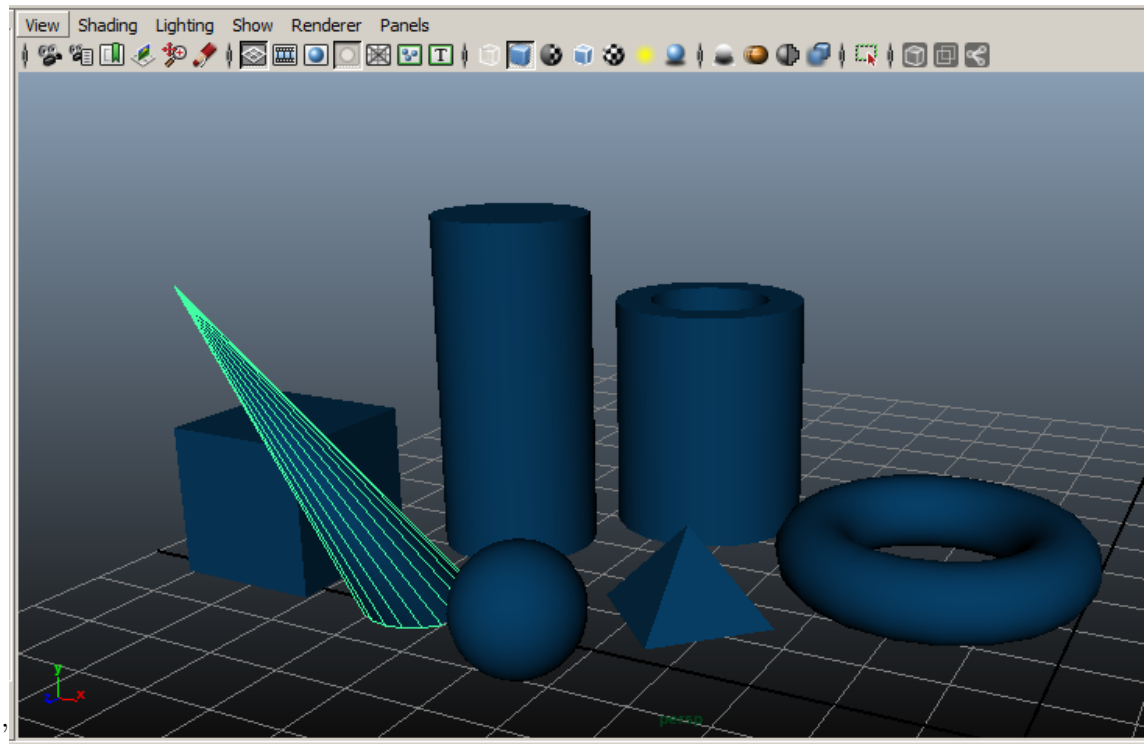


Figure 5.3: The modified cone to be exported in Maya.

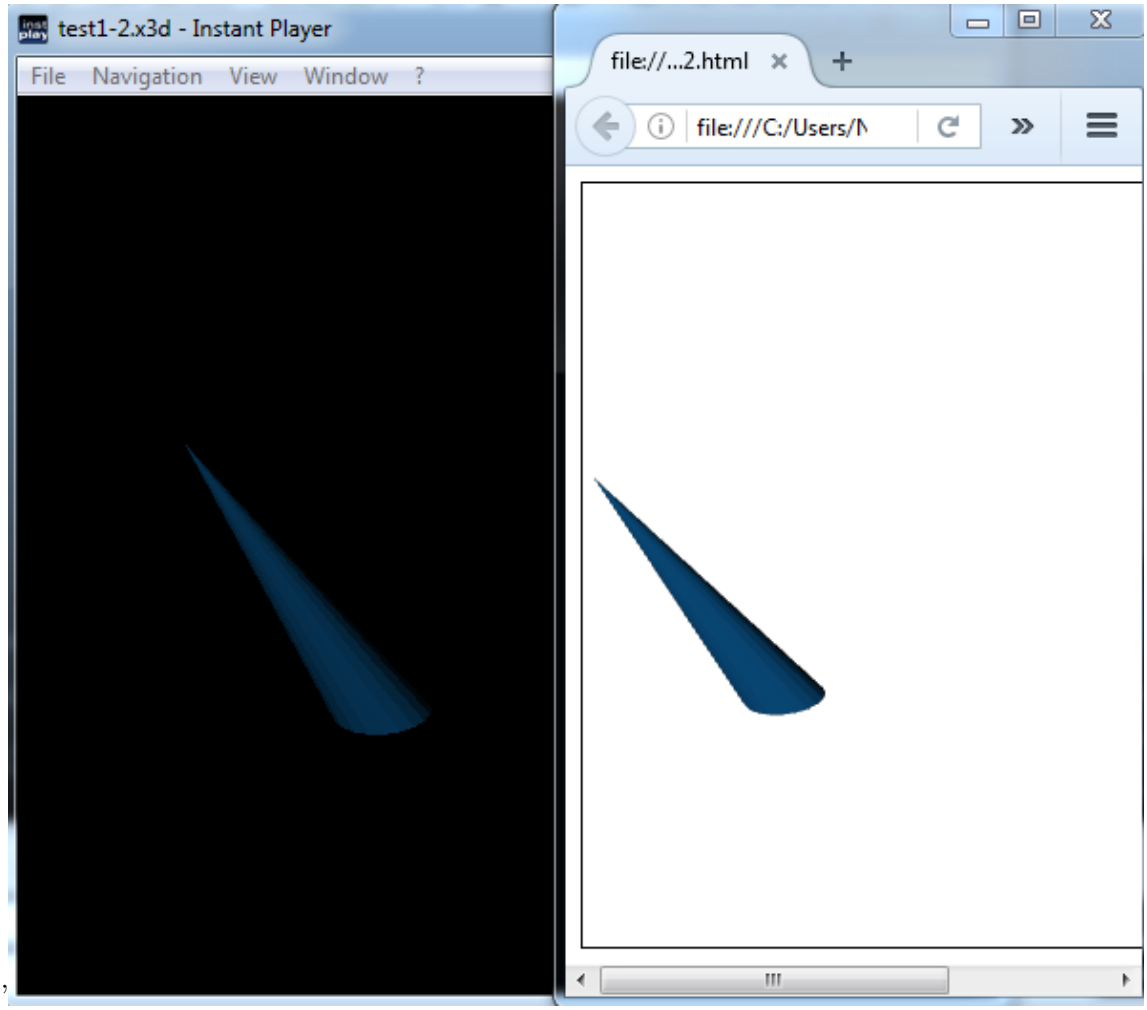


Figure 5.4: The modified basic shape's export results in X3D and X3DOM.

attribute set to the value close to "1" (see Figure 5.5) to X3D and X3DOM with precision set on 4. For this test we use both basic mesh and complex because the texture mapping in X3D for cubes, spheres, cones and cylinders differ from Maya. Figure 5.6 demonstrates the result of the test. As we can see the texture is correctly mapped on the tower and the second tower is quite transparent but on the other hand be cube doesn't have the texture mapped properly.

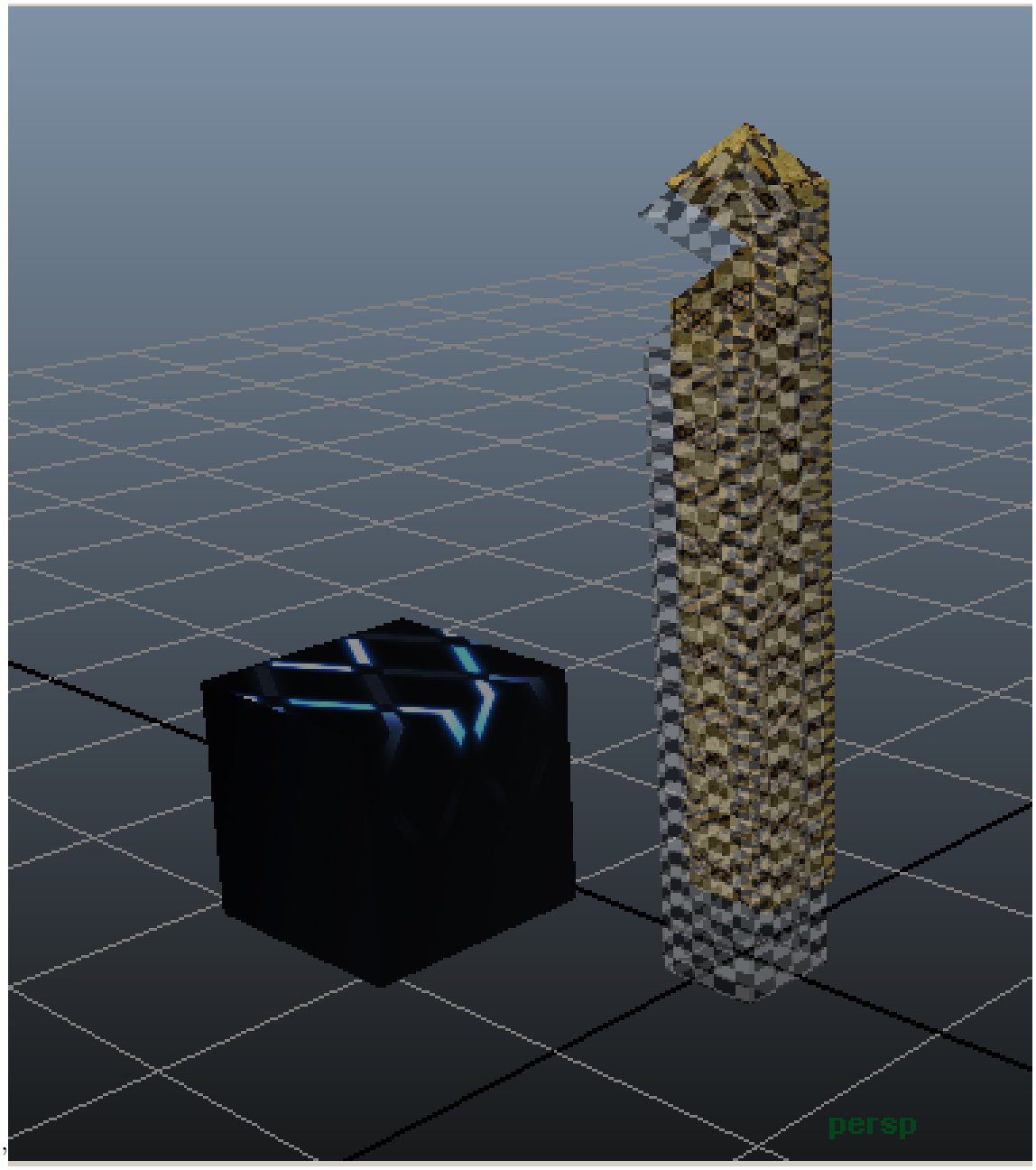


Figure 5.5: A scene with textured objects and a transparent object for appearance testing.

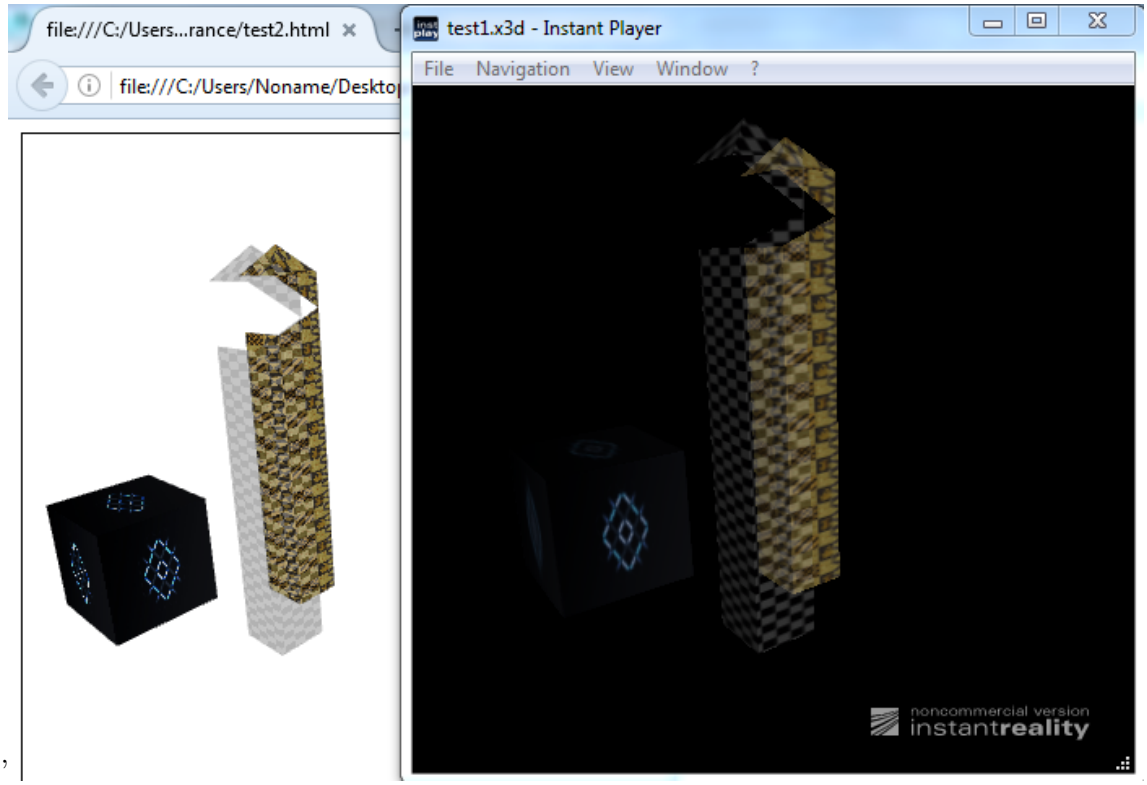


Figure 5.6: The appearance test's results in X3D and X3DOM.

5.1.3 Lights

To make sure the export of the lights works properly we will make the tests below. When comparing the export results we should be sure that the headlights are shut down.

1. **Directional light test** can be conducted on a scene consisting of shapes from the Shape tests and the object of the test itself 5.7. Having exported the whole scene to X3D and X3DOM we want to see the objects in the output files lit by the light as it looks like in Maya. For these test precision is set to "5". The result of X3D export demonstrated on Figure 5.8 is identical to the source scene in Maya but in X3DOM the light's color a little bit differs. However after taking a look at the source files we are sure the export was correct as the color of the shapes' material and the light's color are equal.

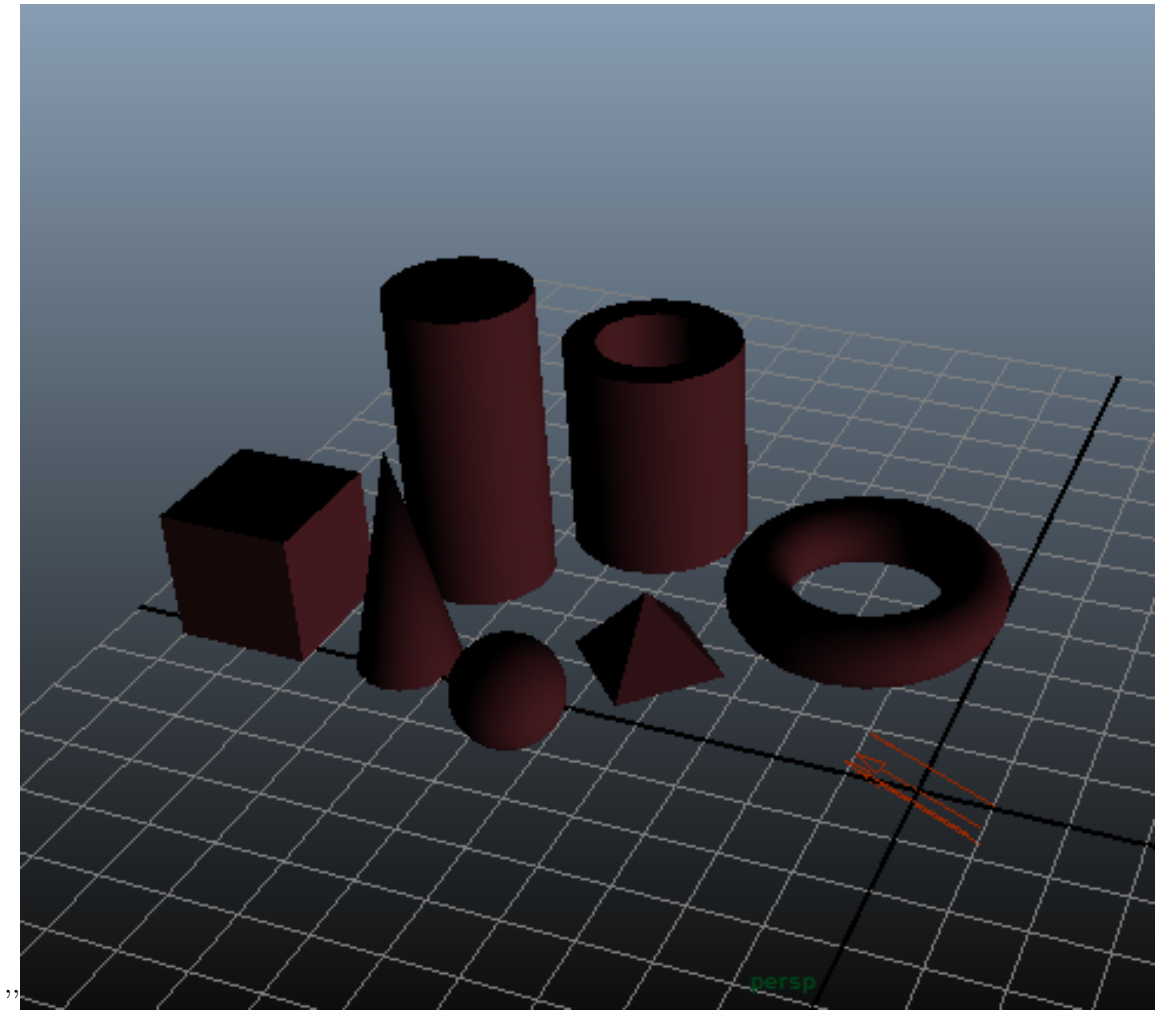


Figure 5.7: A scene with directional light.

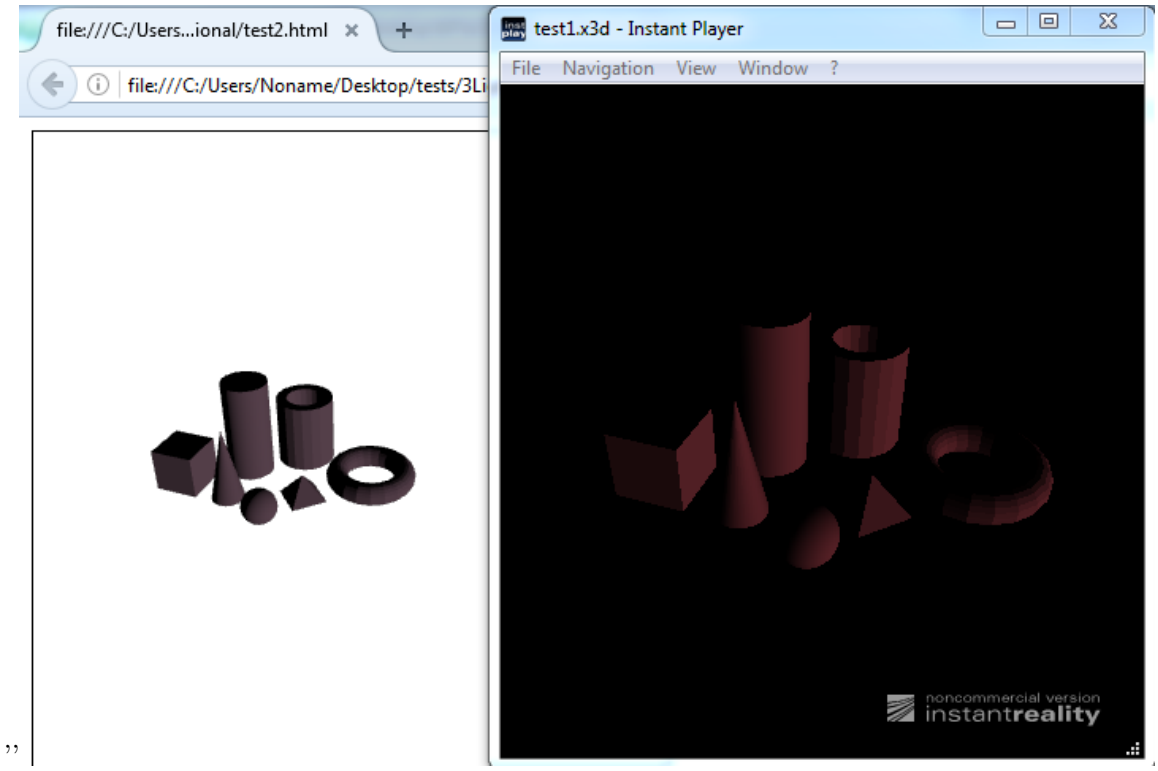


Figure 5.8: The directional light test's result in X3D and X3DOM.

2. **Point light test**. The prepared scene illustrated on Figure 5.9 will be exported to X3D and X3DOM with precision set to "4". We expect that the light will reach the objects from one certain place. The result meets our expectations (see Figure 5.10). In this case the colors are displayed identically. This is caused by the fact that in this test we export the light with white color.

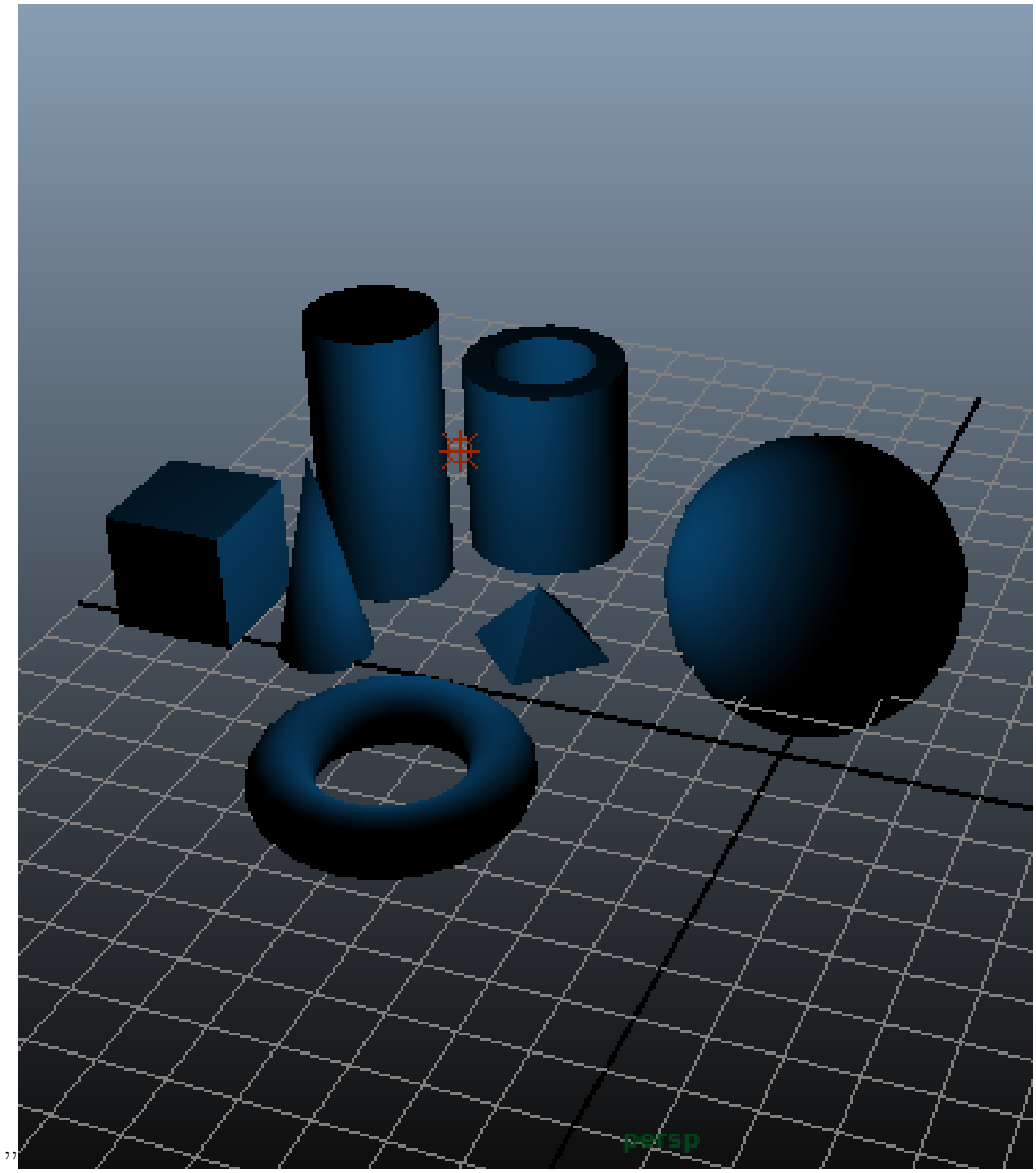


Figure 5.9: The scene with a point light in Maya.

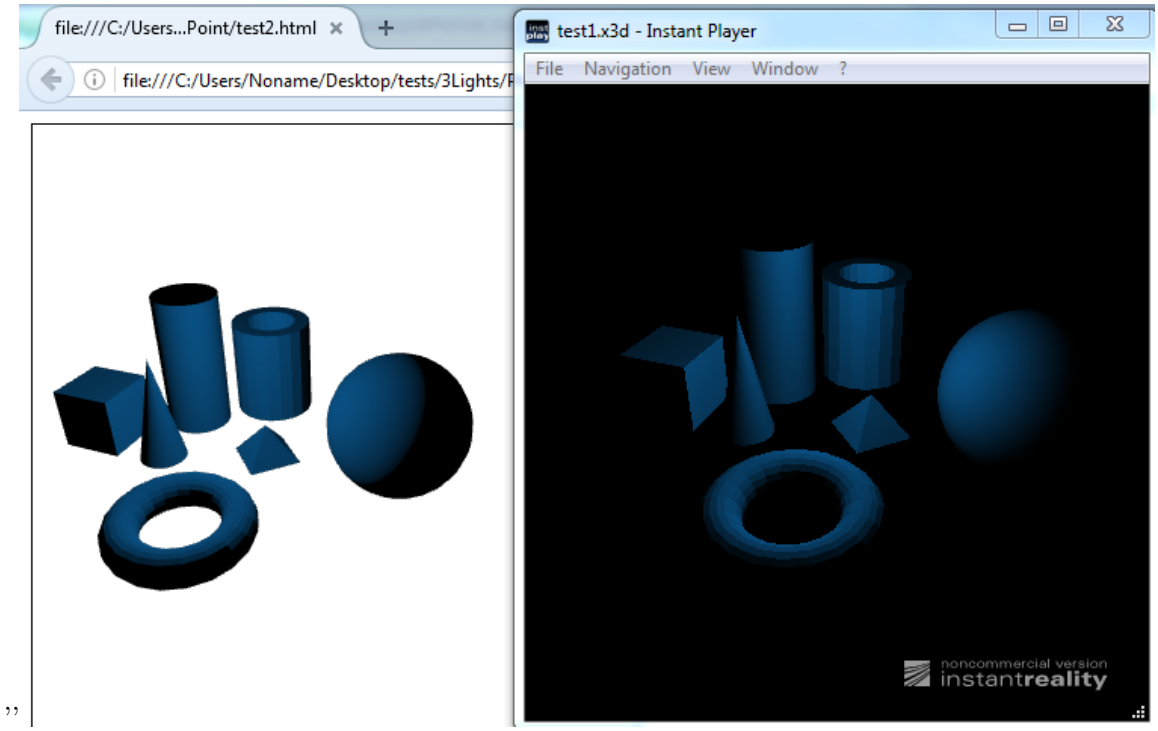


Figure 5.10: The point light test's result in X3D and X3DOM.

3. **Spot light test** by exporting all the objects include the light (see Figure 5.11) to X3D and X3DOM with precision set to "2" and "0" respectively. The light is rotated to make sure the *direction* is being exported correctly. As we can see in Figure 5.12 comparing the results in X3DOM we are shown incorrect results. This is caused by the precision set to "0" - the torus is quite small and with this precision the values in *IndexedFaceSet* are rounded to almost the same small number, the values of the light's *direction* field are also incorrect so we shall not expect the scene to bit lit by the spot light identically to the scene In Maya. If we make another X3DOM test with the precision set to "2" we will get better result.

4. **Spot light test X3DOM** with precision set to "2" demonstrates us on Figure 5.13 that the problem of the incorrect representation of the exported torus is indeed caused by the precision.

5.1.4 Cameras

To make sure the export of the cameras works properly we will make these test:

1. **Export the whole scene with default cameras to X3DOM**. We will set precision to "4" for the export of the scene 5.14. Our expected export result is that we are able to switch *viewpoints* which are correctly exported. The result is in Mozilla Firefox

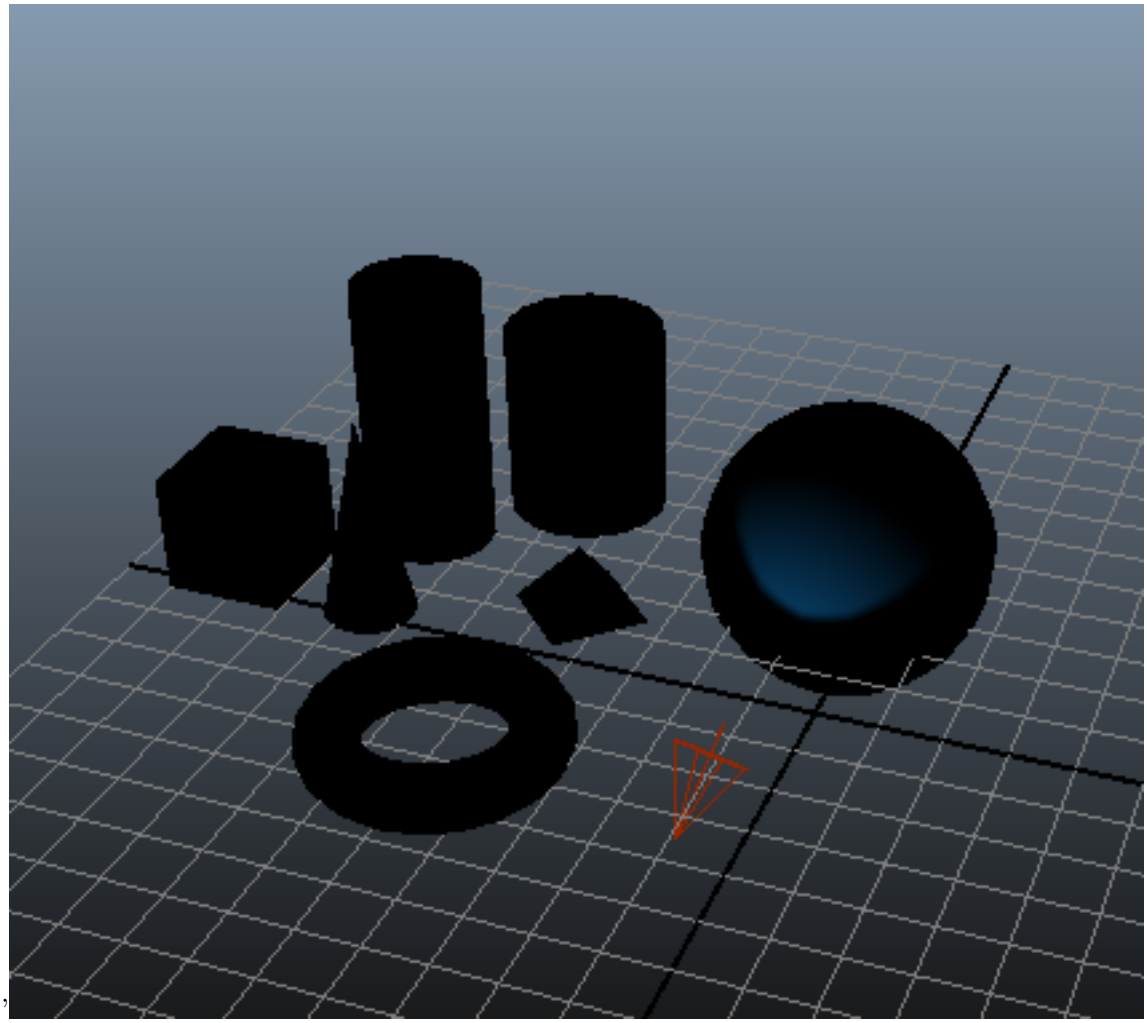
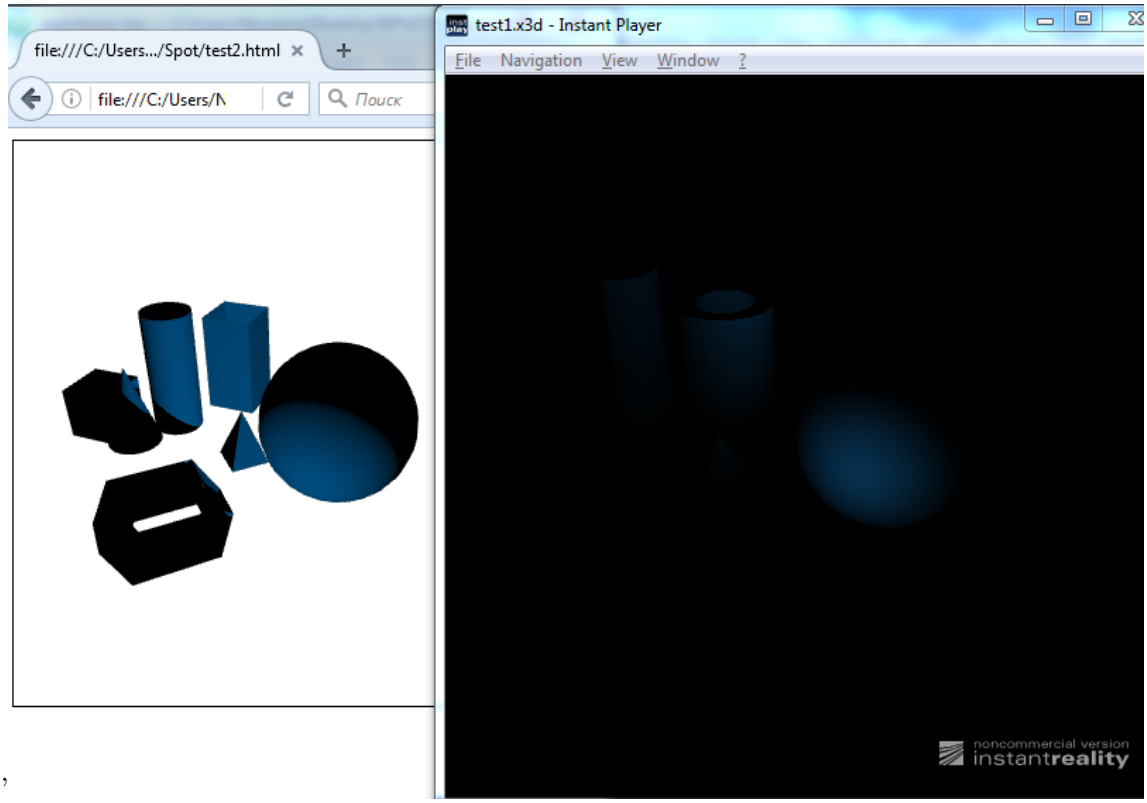
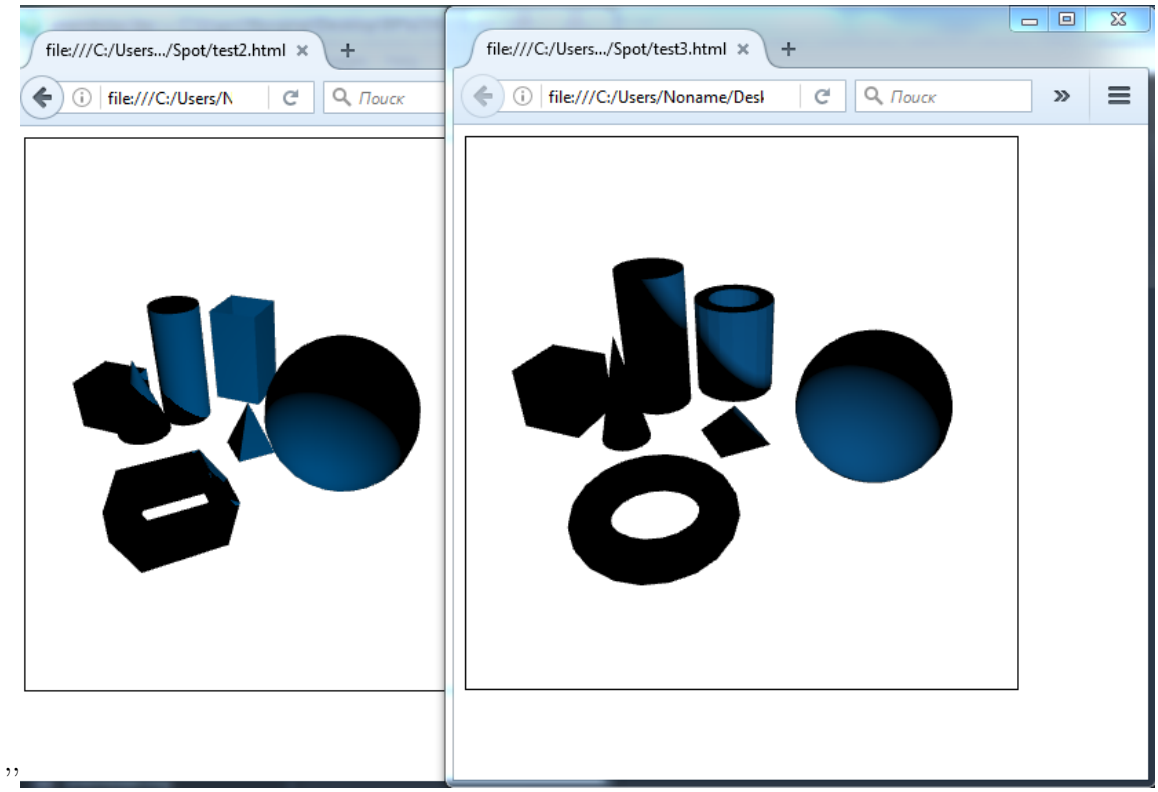


Figure 5.11: The scene with a spot light to be tested in Maya.



,,

Figure 5.12: The spot light test's result in X3D and X3DOM.



,,

Figure 5.13: The spot light test's result in X3D and X3DOM.

we can see the whole scene but we don't gave any possibilities to switch between the *viewpoints*. But if we take a look at the exported file we will find the *Viewpoint* nodes with their attributes' information.

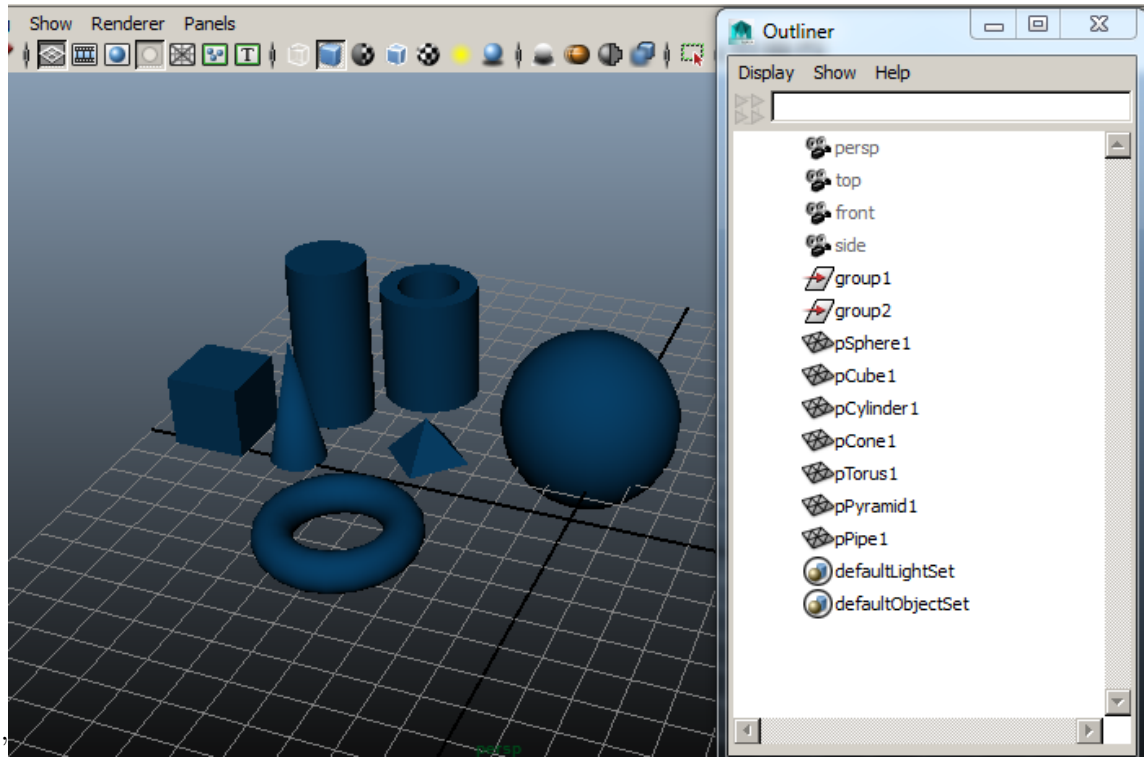


Figure 5.14: The scene which will be exported to X3DOM.

2. **Export selected objects from the scene with default and custom cameras to X3D**. We have created a custom camera called *test* (see Figure 5.15) and are exporting a sphere and a pyramid to X3D having set with precision to the maximum value. We expect all the *viewpoints* to have been exported correctly. To check it we can switch the *viewpoints* in Instant Player (demonstrated on Figure 5.16). Figure 5.17 demonstrates us the result - we can indeed switch the *viewpoints* and the exported *viewpoints* are being exported correctly.

5.1.5 Hierarchical scene

We are going to test exporting a hierarchical scene to X3D and X3DOM with precision set to "6". The hierarchy of the scene is illustrated on Figure 5.18. The result output files (see Figure 5.19) are correctly displayed and stick to the rules of grouping for hierarchical scenes.

5.1.6 Animation

The test **Export the animation** will help us sure we export animation information correctly. The scene 5.20 that we are exporting to X3D and X3DOM with precision set to "3"

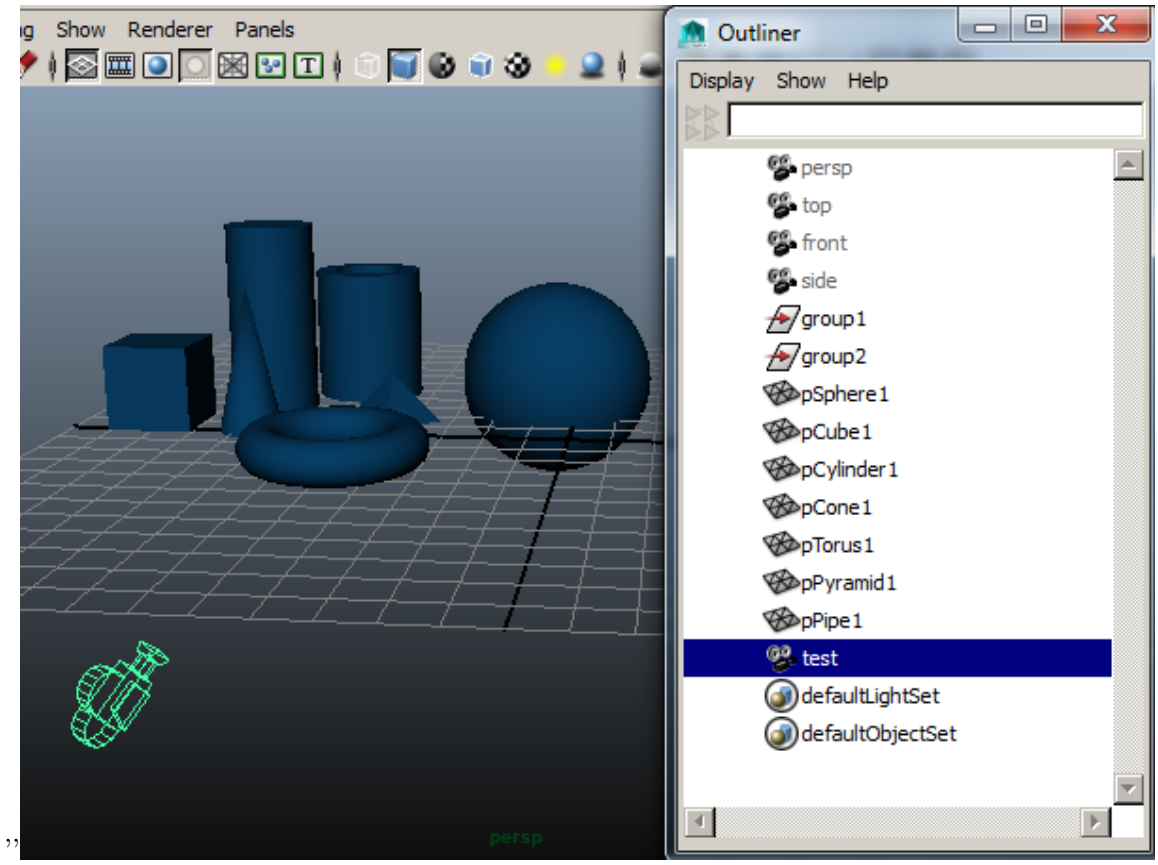


Figure 5.15: The scene with an added custom camera to be exported to X3D.

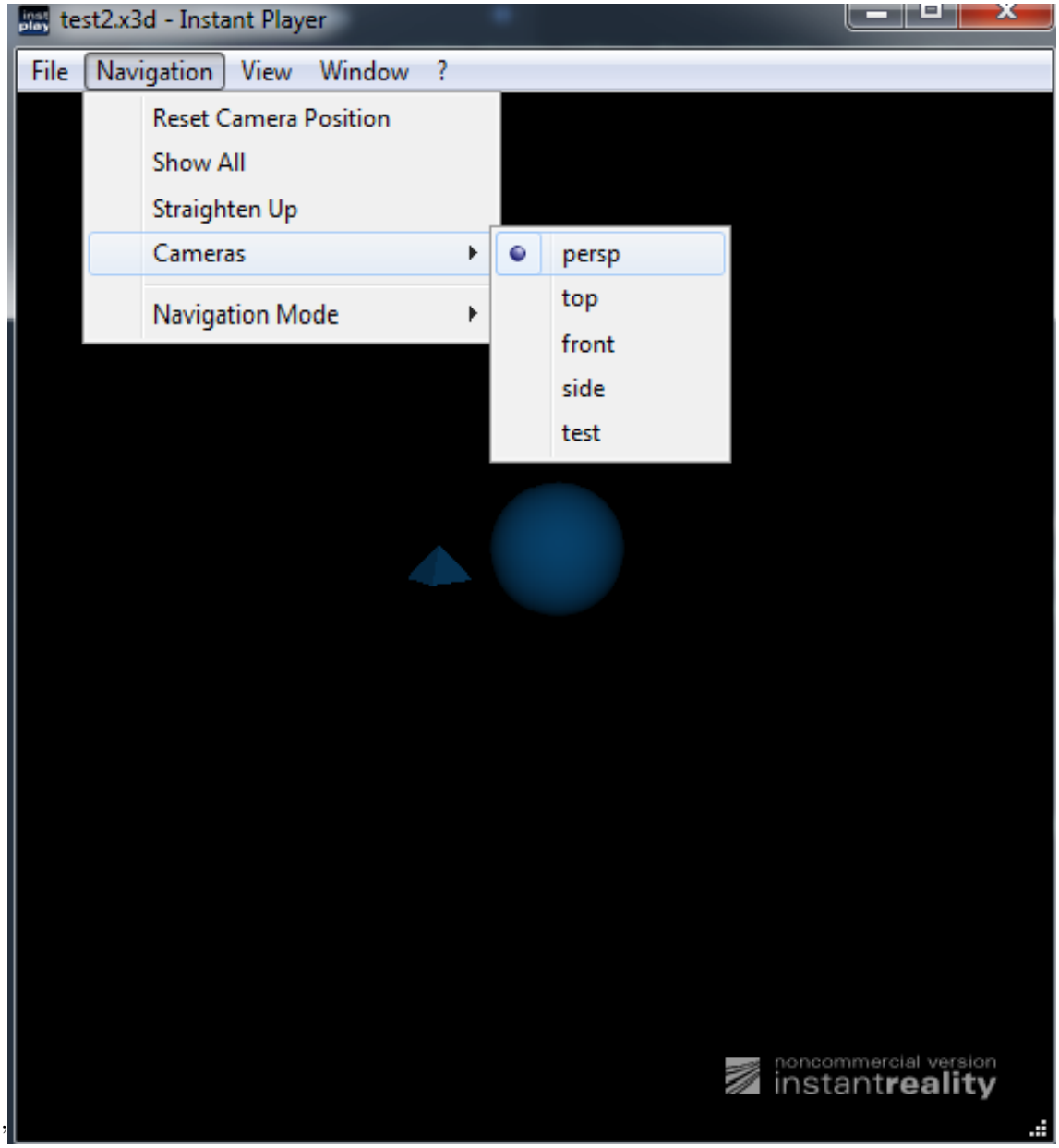


Figure 5.16: Switching *viewpoints* in Instant Player.

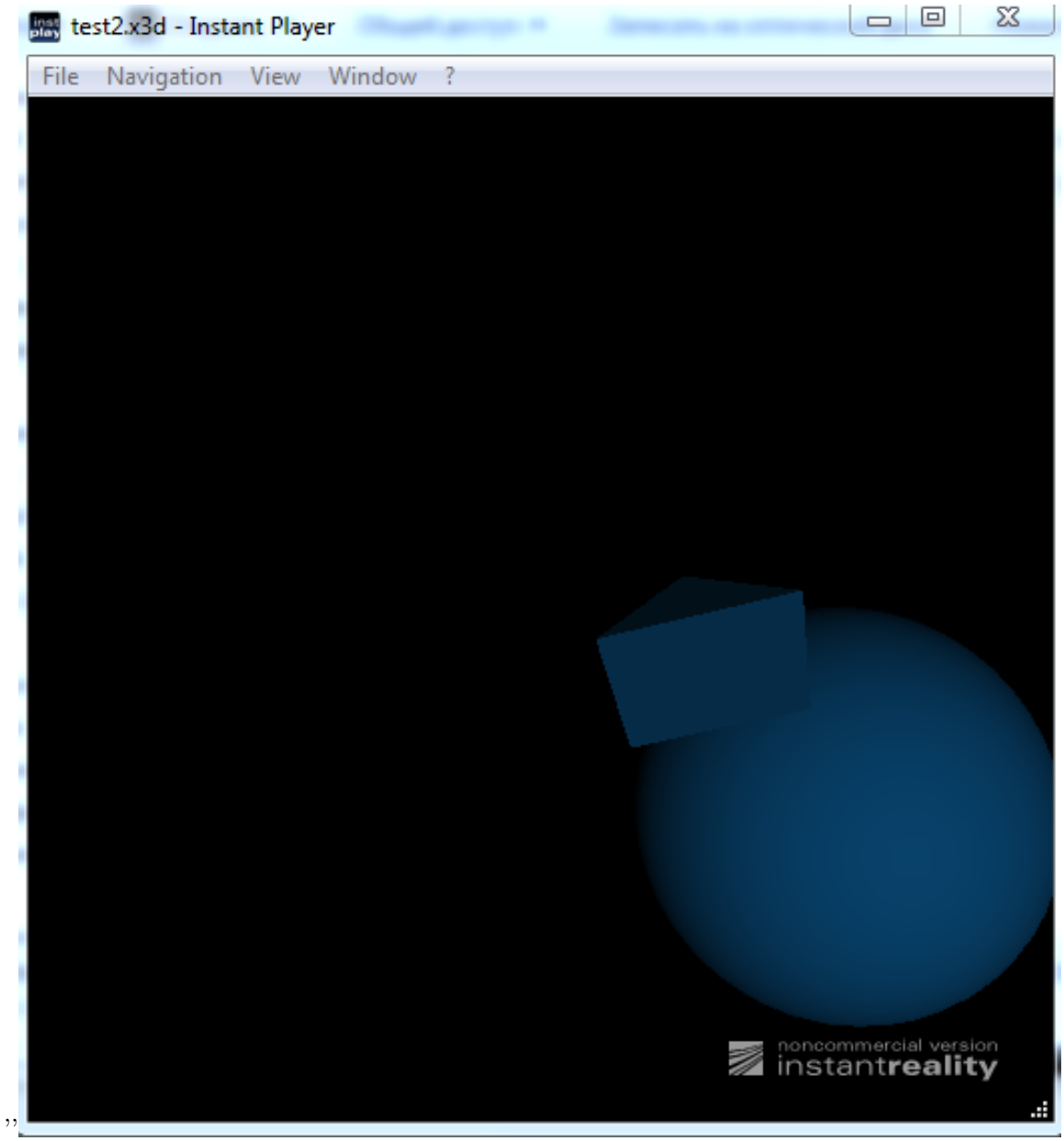


Figure 5.17: The exported objects from the custom camera's *viewpoint*.

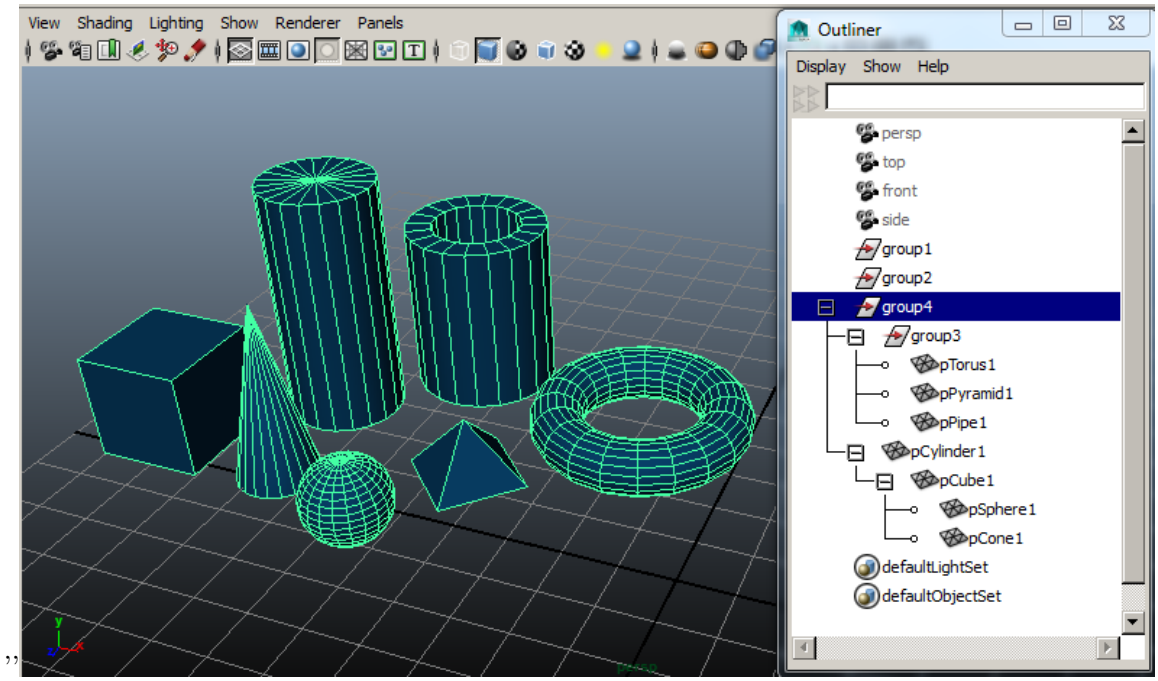


Figure 5.18: The hierarchical scene for the export to X3D/X3DOM.

consists of 5 animated objects - a bouncing and rotating ball and four cylinders that change their scales. We expect the animation works correctly. The result is correct in X3D and X3DOM.

All the test files (Maya scenes and the results in X3D/X3DOM) are available in the *tests.zip*.

5.2 Usability tests

There were chosen five participants between ages 17-43 for the purpose of usability testing. Only one of them had experience with working in Maya.

The participants were given the tasks that can be found in C to test the graphical user interface 3.2 of the plug-in.

Task 1. Export the whole scene to X3D with precision set on the maximum value. *Expectations:* Participants will choose "Export All.." in the File menu. For setting precision will be used the slider or it will be set by typing "6" to the precision field. *Results:* All the participants selected all the objects in the scene, chose "Export All.." and used the slider to set precision on "6".

Task 2. Export the cube to X3DOM with precision set on the minimum value without using the slider. *Expectations:* Participants will choose "Export Selected.." in the File menu and then use the X3DOM radio button. *Results:* The task was completed quickly and correctly as it didn't take time to switch to X3DOM export and the default value of precision is "0".

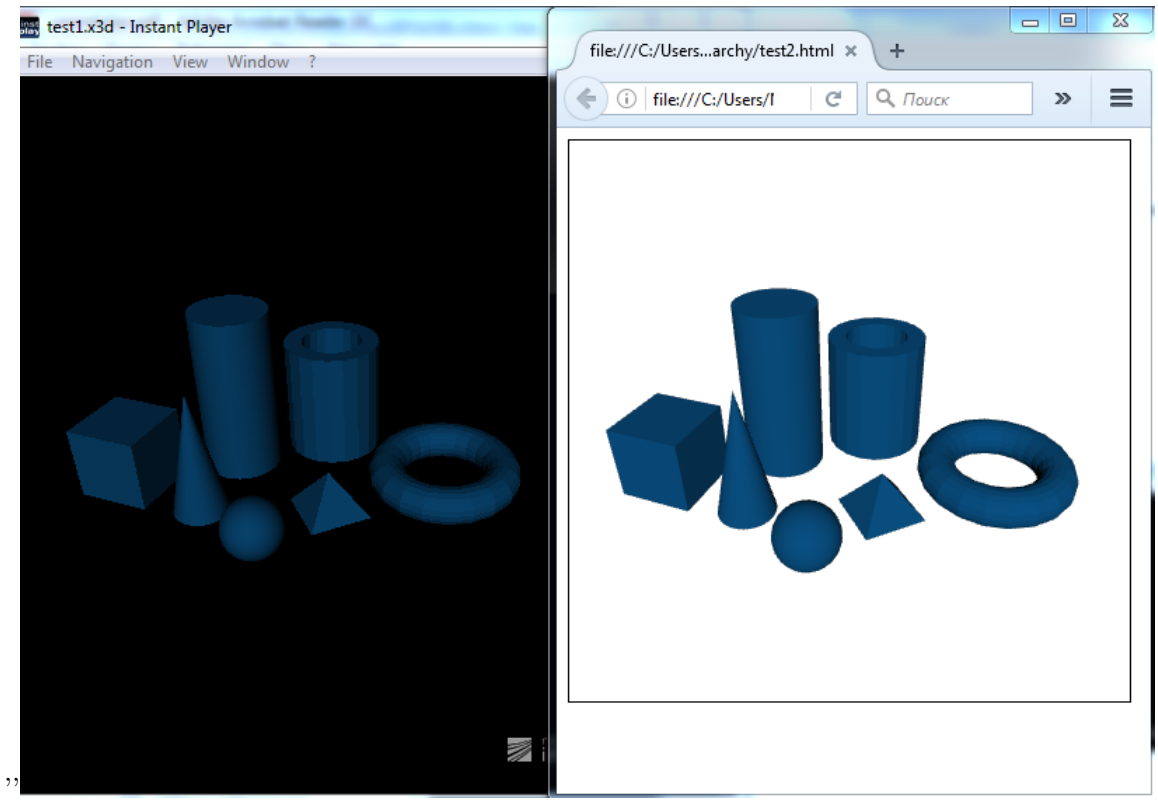


Figure 5.19: The exported hierarchical scene in X3D and X3DOM.

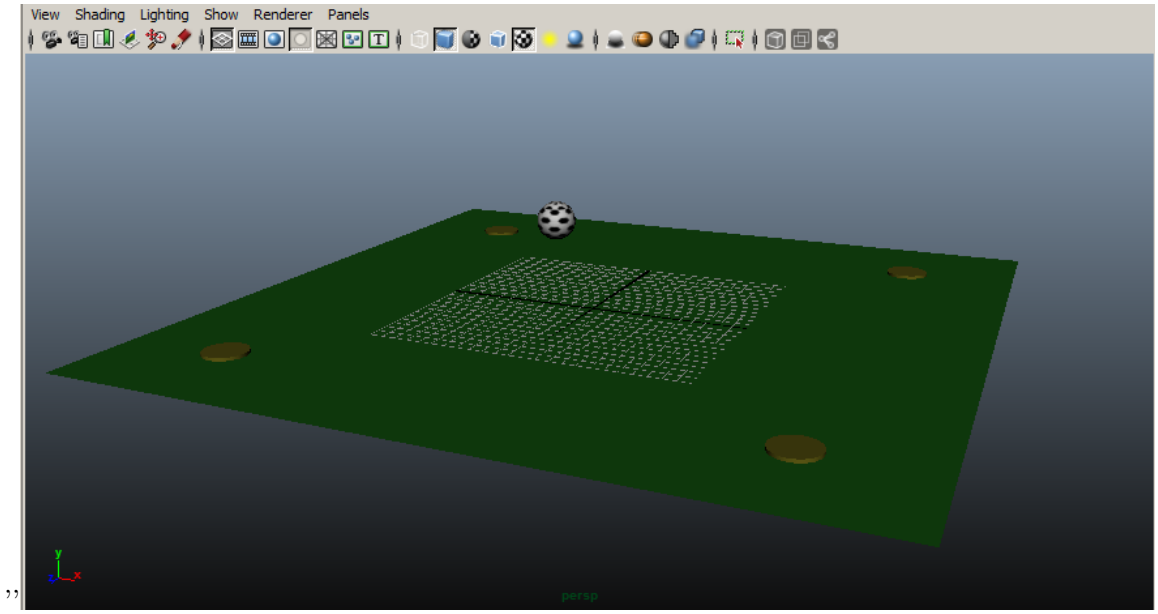


Figure 5.20: The animated scene in Maya.

Task 3. Export the whole scene to X3DOM with precision set on 3 without using the slider. *Expectations:* Participants will choose "Export All.." in the File menu, use the X3DOM radio button and set precision by typing "3" to the precision field. *Results:* Four of the participants selected all the objects in the scene, chose "Export All..", switched to X3DOM and were expected to have some radio buttons for setting precision. After 2-4 minutes each of them tried to type into the precision field. It didn't take time the participant who had had experience with Maya to change precision using the precision field.

Task 4. Export a scene without selecting objects to X3D (using "Export All.."). *Expectations:* Participants will choose "Export All.." in the File menu, fill in the name of the file, successfully export the whole scene to the output file. *Results:* One of the participants expected the "Export All" button remain inactive even if the name of the file is typed in. Four others expected to get an error but not a successful export of the whole scene with the objects that were not even selected.

Task 5. Select nothing and try to export selected to X3DOM (using "Export Selected.."). *Expectations:* Participants will choose "Export Selected.." in the File menu, fill in the name of the file, click on the "Export Selected" button, get the alert "Nothing is currently selected". *Results:* The participants expected a successful export to the output file which will remain empty.

Usability test conclusion is:

- The slider for setting the precision was the best idea,
- Radio buttons for X3D a X3DOM export were chosen as a solution for X3D/X3DOM plug-in correctly,

Chapter 6

Conclusion

In this Bachelor's project I have created an X3D/X3DOM plug-in for Autodesk Maya. The plug-in is capable of exporting with defining values' precision whole Maya scene to X3D format and into HTML file with X3DOM support plugged, specifically all major X3D components (see Table 6.1).

Grouping	switch, group
Geometry	basic and complex shapes
Appearance	materials, textures
Camera	perspective, front, side, top, custom
Lights	point light, spot light, directional light
Animation	position, orientation interpolators

Table 6.1: X3D implemented components.

The only thing that is not implemented yet in Geometry export is capability of plug-in to divide single mesh in partitions given by user.

6.0.1 Future work

In order to improve user experience of this plug-in, amount of export options should be increased. The best source of inspiration will probably be amount of options given in VRML Export plug-in.

Bibliography

[1] Mglobal class reference, 2015. http://help.autodesk.com.

[2] Wavefront .obj file, 2016. https://en.wikipedia.org/wiki/Wavefront_.obj_file.

[3] Vrml, 2017. https://en.wikipedia.org/wiki/VRML.

[4] R. Bateman. Writing a plugin exporter (file translator), 2004. https://nccastaff.bournemouth.ac.uk.

[5] W. Consortium.

[6] D. J. Eck. Introduction to lighting, 2000. http://math.hws.edu/graphicsbook/c4/s1.html.

[7] Fraunhofer-Gesellschaft. - x3dom.org, 1999. http://www.x3dom.org/.

[8] I. W. P. H. F. L. GABRIEL TAUBIN, SENIOR MEMBER and J. ROSSIGNAC. Geometry Coding and VRML, 2002. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=687837&tag=1.

[9] D. Goulda. *Complete Maya Programming, 1st Edition An Extensive Guide to MEL and C++ API*, volume 1. Elsevier Science, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, 1st edition edition, 2003.

[10] N. Mead. Powerful 3d modeling, animation and rendering solution, 2014. http://autodesk-maya.en.softonic.com/mac.

[11] C. Vernon. Maya API Programming. http://www.chadvernon.com/blog.

Appendix A

Attached files

- **exporter.zip** - contains the source code and the .mll plug-in file (inside the *Release* folder)
- **MELscript.zip** - contains a MEL script which should be located in a user's `maya/scripts` folder
- **thesis** - contains thesis pdf file
- **tests.zip** - contains the result test files and the files used for the testing purposes
- **documentation** - contains documentation
- **imgs.zip** - contains images for propagation of FEL CVUT

Appendix B

Installing the plug-in

The plug-in should be loaded into Maya's *Plug-in manager* by following these steps:

- open *Window menu*
- go to *Plug-in manager*
- browse the .mll file from the plug-in's *Release* folder

The MEL script must be located in the `\maya\scripts` folder. The needed Maya version is 2015.

Appendix C

Tasks for the usability testing

1. Export the whole scene to X3D with precision set on the maximum value.
2. Export the cube to X3DOM with precision set on the minimum value without using the slider.
3. Export the whole scene to X3DOM with precision set on 3 without using the slider.
4. Export an empty scene (with no selected objects) to X3D.
5. Select nothing and try to export selected to X3DOM.

Appendix D

Other references

The information about the plug-in and demonstrations of its work are available at: https://sites.google.com/site/x3dexportplugin/.