Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Interaction of a computing environment with a 3D scene

Tereza Hlavová

Supervisor: Ing. Jan Houška
Field of study: Open Informatics
Subfield: Computer Games and Graphics
May 2023

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Hlavová Tereza**	Personal ID number:	**492278**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Graphics and Interaction**		
Study program:	**Open Informatics**		
Specialisation:	**Computer Games and Graphics**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Interaction of the MATLAB computing environment with 3D scene

Bachelor’s thesis title in Czech:

Interakce výpočetního prostředí MATLAB s 3D scénou

Guidelines:

Familiarize with the current state of open formats for 3D scene description, with an emphasis on formats that are internationally standardized. Compare available open-source implementations of these formats in terms of rendering quality and speed, availability of advanced features (collision detection, reflections, physics, ...), and licensing terms. Explore the possibilities of two-way transfer of information (data and events) between 3D scene and another environment. Focus on communication with the MATLAB computing environment and the Simulink simulation tool. Investigate the current implementation of 3D scene handling in the Simulink 3D Animation tool and suggest modifications to better support open standards and increase rendering quality and speed. Implement the suggested modifications and evaluate their impact on rendering performance. To evaluate rendering quality and performance improvements, use the shipping examples from the Simulink 3D Animation tool and other virtual scenes supplied by thesis supervisor.

Bibliography / sources:

S. Marschner and P. Shirley: Fundamentals of Computer Graphics, 4th edition, CRC Press, 2016.
M. Pharr, W. Jakob, G. Humphreys: Physically Based Rendering: From Theory to Implementation, 3rd edition, publicly online at https://pbr-book.org
Extensible 3D (X3D) version 3.3, ISO/IEC 19775-1:2013, https://www.web3d.org/documents/specifications/19775-1/V3.3
Extensible 3D (X3D) version 4.0, Committee Draft, https://www.web3d.org/documents/specifications/19775-1/V4.0
glTF™ 2.0 Specification https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html
x3dom https://doc.x3dom.org
X_ITE X3D Browser https://create3000.github.io/x_ite
Bullet Collision Detection & Physics Library https://pybullet.org/Bullet/BulletFull/annotated.html

Name and workplace of bachelor’s thesis supervisor:

Ing. Jan Houška HUMUSOFT s.r.o.

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **17.02.2023** Deadline for bachelor thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

Ing. Jan Houška Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce her thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

I want to thank my family and friends for support during my studies and my supervisor Ing. Jan Houška for consultations and advice for this work.

Declaration

I declare that I have made this work on my own and that I have specified all used literature.

24. 5. 2023, Prague.

Abstract

In the field of 3D graphics, various formats describing the 3D scene have been created and various renderers capable of working with such formats have been produced, all with different capabilities in terms of rendering quality, object physics and performance.

The goal of this work is to examine current internationally standardized formats describing the 3D scene, compare 3D graphics libraries that can visualize the scene with an emphasis on their use in a MATLAB tool Simulink 3D Animation and overview the current state of the software implementation, then provide an outline of changes to the software which would lead to overall better visual quality, object physics simulation capabilities and speed, and implement proposed modifications.

Keywords: 3D scene, physical simulation, physically based rendering, X3DOM, X_ite, Three.js, Simulink 3D Animation

Supervisor: Ing. Jan Houška

Abstrakt

V oboru 3D grafiky vzniklo nespočet různých formátů pro popis 3D scény a podobně tak i mnoho 3D grafických knihoven s nimi pracujících, jejichž schopnosti s ohledem na kvalitu vykresleného obrazu, fyziku objektů a výkon jsou různorodé.

Cílem této práce je prostudovat existující mezinárodně standardizované formáty pro popis 3D scény, porovnat 3D grafické knihovny, které dokáží takto reprezentovanou scénu zobrazit, s důrazem na jejich možné použití v nástroji Simulink 3D Animation pro software MATLAB, nastudovat nynější implementaci tohoto nástroje, poté poskytnout návrh jeho úprav pro zlepšení visuální kvality, schopnosti simulace fyziky objektů a vyšší rychlost, a následně navržené změny implementovat.

Klíčová slova: 3D scéna, fyzikální simulace, physically based rendering, X3DOM, X_ite, Three.js, Simulink 3D Animation

Překlad názvu: Interakce výpočetního prostředí s 3D scénou

Contents

Figures

Tables

Chapter 1

Introduction

In the last 50 years or so, 3D graphics have come a very long way with new breathtaking improvements in visuals promised every year. It is almost hard to keep up with, as with most branches of IT. Though one trend has still stayed and that is a desire for achieving photo-realism in rendering. Apart from different directions being chosen for artistic reasons, photo-realistic shading/illumination, realistically behaving physics and collisions are wanted even for highly stylized objects in the 3D scene. A decent level of photo-realism has already been achieved, but there is a catch and that would be achieving such results in a reasonable time.

The goal of this work is to look at existing file formats describing the 3D scene that are standardized worldwide. Then to go over 3D graphics libraries capable of working with these file formats and compare their rendering quality and speed, collision detection and object physics implementation, their license types and their capabilities in terms of communication with an external software, mainly with the computing environment MATLAB and simulation tool Simulink. Then, after studying the current implementation of a tool Simulink 3D Animation, which can be seen in figure 1.1, depending on the results of the 3D graphics libraries comparison, and also while taking the current state of the tool into account, changes to the software enhancing its rendering quality and performance will be proposed and implemented.

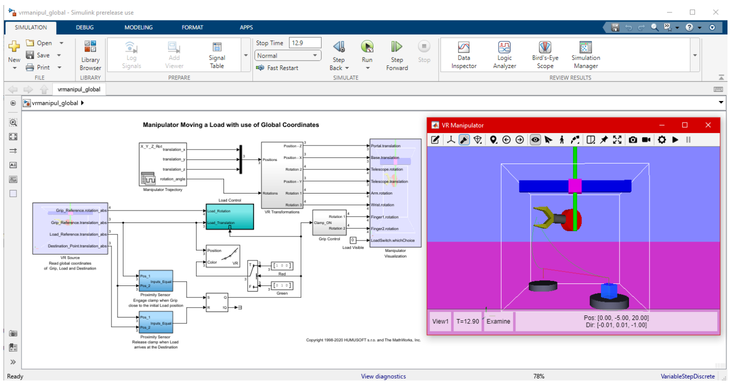

vrmanipul_global - Simulink prerelease use
SIMULATION
DEBUG
MODELING
FORMAT
APPS
Manipulator Moving a Load with use of Global Coordinates
VR Manipulator
View1
T=12.90
Examine
Pos: [0.00, -5.00, 20.00]
Dir: [-0.01, 0.01, -1.00]
Ready
View diagnostics
78%
VariableStepDiscrete

Figure 1.1: Simulink 3D Animation, a tool of MATLAB and Simulink, with a newly implemented renderer, used to visualize a simulation in a 3D scene.

This work is divided into 7 chapters. Chapter 2 introduces the file formats and their standards. Chapter 3 deals with the comparison between chosen 3D graphics libraries. Then, chapter 4 explores the current implementation of Simulink 3D Animation and in chapter 5 modifications to it are proposed and their implementation is described. The results are presented in chapter 6. Finally, chapter 7 summarizes all work done and proposes future work on the project.

Chapter 2

Standardized Formats for 3D Scene Representation

This chapter will go over format specifications for 3D scene representation. For long-term usability and compatibility, it will be focused on format specifications standardized by the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC), which currently are VRML, X3D and glTF formats.

2.1 VRML

VRML, a shortened term for The Virtual Reality Modeling Language (before 1995 known as Virtual Reality Markup Language), is a declarative file format describing 3D objects and scenes. Its design allows a 3D content editing software to implement tools for the creation and editing of VRML files, to combine dynamic 3D objects of predefined or new object types and to compose them in a 3D scene. It also makes compatibility with a wide range of systems and working with large scenes possible.[1]

VRML was designed by the Web3D Consortium, previously known as the VRML Consortium. The first version of the specification was finished in 1994. It was then followed by VRML2.0 in 1996 with its draft submitted for standardization. Next year, Part 1 of VRML97 was accepted as an international standard by ISO/IEC. Part 2 of the specification that focuses on describing communication of an external environment with the VRML scene through the VRML event model became part of the standard in 2004.

2.1.1 Concepts

A VRML file represents a 3D scene and its behavior through a header, a scene graph and event routing. The scene graph is a directed acyclic graph of nodes. It describes a transformation hierarchy and specifies its behavior under an event system.

A node is an abstraction of a real-life object or concept. It contains a type name, a set of events it can generate or receive, an implementation, and a set of properties and attributes called fields, plus optionally also a name. A field can be so-called single-valued, containing a single value, such as a numeric value, whole image or even another node. Multi-valued fields contain an ordered list of values. If a node's field contains another node or nodes, a descendant and ancestor relationship is created between them.

VRML files are to be viewed by a VRML browser which is a computer program capable of presenting their content and allowing interaction with the 3D scenes through UI, user interface.

An example of how a red ball object, scaled up uniformly to the value of 1.5, and positioned 20 base units on a Y axis of the global coordinate system, would be represented in a VRML file, can be seen on the following snippet:

```
DEF Ball Transform {
  translation 0 20 0
  scale 1.5 1.5 1.5
  children Shape {
    appearance Appearance {
      material DEF Ball_material Material {
        diffuseColor 1 0 0
      }
    }
    geometry Sphere {
    }
  }
}
```

Prototypes

Although VRML specification offers a wide range of built-in node types, user is able to define new node types through prototypes. A prototype is defined by its own fields and one or more nodes, which are copied and inserted in all places where the prototype is instantiated in the scene, with fields filled by either default values or those passed to it upon instantiation.

DEF/USE Statements

A **DEF** keyword may be used to give a node a name, giving a user a tool for reusing the node with a **USE** keyword or referencing the node in event routing. When a **USE** keyword with an already defined name is present in the scene graph, the original node with the **DEF** keyword is not copied, but rather gains one more ancestor and is reinserted at a new place in the scene graph.

Grouping nodes

A node that contains a field of multiple so-called children nodes is considered a grouping node and a parent to all its children nodes if such field is non-empty.

Grouping nodes partake in the creation of a transformation hierarchy of the 3D scene, which affects most of the node types of the VRML specification. Built-in grouping nodes of VRML specification are: **Anchor**, **Billboard**, **Collision**, **Group**, **Inline**, **LOD**, **Switch** and **Transform** nodes.

A grouping node's transformation defines a local coordinate system of its children, and does so relative to its ancestors' coordinate systems. A 3D scene is usually built from various combinations of grouping nodes, mainly **Transform** nodes, which provide the user the ability to directly modify the transformation matrix applied to its children through its attributes of *translation*, *rotation* and *scale*.

Light sources

VRML standard defines **PointLight**, **SpotLight** and **DirectionalLight** nodes. They too are influenced by the transformation hierarchy and thus their lighting properties are considered in their local coordinate systems.

Shapes and geometry

Virtual 3D objects are mostly represented by so-called **Shape** nodes. They would usually contain a node describing the geometry of the object and an **Appearance** node, under which a **Material** node and/or texture node would be described.

VRML standard offers the following geometry nodes: **Box**, **Cone**, **Cylinder**, **ElevationGrid**, **Extrusion**, **IndexedFaceSet**, **IndexedLineSet**, **PointSet**, **Sphere** and **Text** node.

Defined texture nodes - **ImageTexture**, **PixelTexture** and **MovieTexture** nodes are applied to the shape based on a **TextureTransform** node if specified under **Appearance**.

VRML standard then specifies how the finished object's appearance should be rendered, taking its geometry, its appearance, and surrounding light nodes in the scene into account.

Viewpoints

Viewpoint nodes in the scene are influenced by the transformation hierarchy as well. When a **Viewpoint** node is *bound*, a scene camera gets parented by it and the node defines the camera's local coordinate system in which the camera's movement may be realized.

Events

Events can serve as a link of nodes with other nodes making communication through messages via routes possible. Nodes can create and receive events if such behavior is specified for the node type.

Events can be also generated and received by **Script** nodes, which are not a part of the transformation hierarchy but can indirectly influence it through the events.

The event model and delivery of events should be handled by the VRML browser.

Sensors

Interaction with the user can be enabled through sensor nodes. Upon user's clicks, device movement or even movement in space, events are produced by the sensors for further processing in other nodes or scripts.

2.1.2 Rendering

VRML specification does not include the name of the lighting model used, but what it does describe corresponds to Blinn-Phong lighting model almost perfectly with minimal deviations. VRML does not specify anything about shadow casting or shading.

VRML does not offer an in-built solution for normal mapping or any other mapping outside of diffuse texturing.

2.1.3 Collision detection and physics

VRML describes only very basic collision detection between objects and the viewer's avatar. Upon such collision, the avatar's position should be influenced accurately. Terrain following implementation is hinted at there, but there is no mention of rigid body physics or even collision detection between defined objects.

2.2 X3D3.3

VRML standard has since been superseded by X3D, a short form for "Extensible 3D", and stays as a direct subset of X3D. The latest currently standardized version is version 3.3, which was finished in 2015. It improves upon VRML with new features, advanced application programmer interfaces, additional data encoding formats, stricter conformance, and a componentized architecture that allows for a modular approach to supporting the standard.[2]

X3D introduces the concept of abstract nodes, seemingly simplifying node type definition by creating an inheritance hierarchy. Thus, creating new node types is made easier when one can derive them from abstract nodes.

X3D specification defines distinct sub-specifications called **Profiles**. Choosing one of them, an application or an X3D browser can limit which features it supports.

As previously shown with the VRML file format, the same example red ball object, now represented by X3D XML file format, can be seen on the snippet below:

```
<Transform DEF='Ball'  translation='0 20 0' scale='1.5 1.5 1.5'>
  <Shape>
    <Appearance>
      <Material DEF='Ball_material'  diffuseColor='1 0 0'>
      </Material>
    </Appearance>
    <Sphere>
    </Sphere>
  </Shape>
</Transform>
```

2.2.1 Rendering

Like in VRML, the illumination model described for lighting objects is again Blinn-Phong model. There is still no mention of casting shadows. X3D 3.3 does however support multiple textures on one object with blending specifications needed. This does allow the use of pre-calculated lightmaps. However, a direct specification of normal maps or textures for specular reflection is still not included. Their use can be achieved with a custom shader node that for example X3DOM specifies and offers to its users.

2.2.2 Collision detection and physics

In the X3D 3.3 specification, a whole new component **Rigid body physics** is dedicated to collision detection and object physics. A physics model, if used, is added to the execution model of the X3D browser at the end of every frame.

To ensure the physical simulation works correctly, additional nodes must be provided. Apart from the visual itself contained within a **Shape** node, **CollidableShape** node is needed as well as a **RigidBody** node.

CollidableShape node should serve as a link between the simulation and the rendering. It contains a geometry proxy, a **Shape** node. The **Shape** node should represent the colliding object and allow the physics model to move it.

In the *geometry* field, **RigidBody** node describes attributes of its **CollidableShape** that are needed for simulations of the physics model. Most important is mass, whether to use gravity or not, damping factor, whether it is fixed or not, which forces it is influenced by, and what are its starting properties of linear and angular velocity. **RigidBody** nodes should be placed inside a **RigidBodyCollection** node, the content of which is not rendered. It is implied that the geometry used for physics might be less detailed than the one used for rendering. The **RigidBodyCollection** node may also include joints.

Joints link together two bodies and create constraints in their movement relative to each other. Each joint usually has a possible range of motion defined in terms of linear distance and radial angles.

Collision detection is also used in a **Picking Component** and **Pointing Device Component** which both specify various sensor nodes with ray and shape casting abilities. Most notable for this work is a **LinePickSensor** node from the **Picking Component**. It is to be inserted in the scene graph with line geometry defining the rays to be cast and a shape or a grouping node with which the intersections of the rays shall be tested. Every time its segment does intersect with the given group, it generates and *isActive* event with additional information, mainly the nodes it hit and the point in space it hit in.

2.3 X3D4.0

New X3D standard version 4.0 currently exists as a draft. It is under review to be added to international standards as well. Although at this moment its specifications might not yet be stable, its additions and changes might be crucial for this and future work.

2.3.1 Theory

X3D 4.0 includes more lighting models than the previous versions. Possibly the most important addition is the inclusion of physically based rendering, PBR.

It also introduces image-based lighting and shadow casting.

Physically based rendering

Physically based rendering describes a method of shading and rendering that aims to represent an interaction between light and a surface of an object more accurately than empirical local illumination methods. [4] Under such model, objects with defined materials should be rendered to look consistent under any lighting setup in the scene.

For a lighting model to be considered physically based, three conditions have to be met:

- the model must comply with the energy conservation law
- it must be based on the microfacet theory
- its used BRDF, Bidirectional Reflectance Distribution Function, must also be physically based.

For every point of an object's surface, the BRDF takes as parameters the direction of incoming light, the direction of the camera in the scene, the normal of the surface in the given point, and material parameters in the point - the roughness of the surface.

Outgoing light from the object's surface must never exceed the amount of light that illuminates it, except for light emission. Upon hitting the surface, light can reflect, refract, or do a combination of both. Refracted part of the light is either absorbed or will scatter, resulting in getting absorbed later, re-emerging through the surface, or passing through the object and re-emerging there.

Re-emerged light defines what's considered a diffuse color. PBR introduces a metallic parameter, which approximates the surface's behavior based on the conductor-insulant differentiation. Metallic objects absorb all of the refracted light.

One of the PBR illumination models is Cook-Torrence illumination model, which uses the following reflectance equation:

$$\vec{C} = k_d\vec{C}_D + k_s\vec{C}_{Cook-Torrence} \tag{2.1}$$

$$\vec{C}_D = \frac{\vec{C}_b}{\pi}\vec{C}_L(\vec{L}\vec{N}) \tag{2.2}$$

$$\vec{C}_{Cook-Torrence} = \frac{1}{\pi}\vec{C}_L\frac{DGF}{(\vec{V}\vec{N})(\vec{L}\vec{N})} \tag{2.3}$$

$\vec{C}$, a color resulting from a ray traced from one light source, is a sum of refracted diffuse color, $\vec{C}_D$, and reflected specular color, $\vec{C}_{Cook-Torrence}$. Sum of parameters k_d and k_s cannot be greater than 1. $\vec{C}_b$ is the base color of the material, $\vec{C}_L$ represents the color of the light, $vecV$ and $\vec{L}$ are the directions of view vector and light ray and $\vec{N}$ is the normal vector.

D stands for a distribution function, which models the roughness of the surface, G is a geometry function which modeling shadowing and masking of surface microfacets and F is a Fresnel function which models the Fresnel effect.

Image based lighting

Image-based lighting, or IBL for short, is a method of illuminating the 3D scene which treats an environment cube texture map as a light source. It usually specifies both diffuse indirect lighting and specular indirect lighting for the scene.

The diffuse indirect lighting map is usually pre-computed with the use of a convolution method. During the computation of the reflectance equation, this map gets sampled as an environment map and its sampled point is treated as a light source specifying the diffuse portion of incoming light.

Meanwhile, the indirect specular map gets applied upon computation of a reflected eye vector. The value of the indirect specular map at the point of direction of this reflected vector is then treated as a specular light source.

Shadow casting

Cast shadows in the scene are a result of a light ray being occluded by an object from reaching the surface of another object. A very common technique for rendering shadows is shadow mapping

The idea behind shadow mapping is to have a so-called shadow camera in place of a light source - an orthographic camera for directional light, a perspective camera for a spotlight, and 6 perspective cameras for point light. During a so-called shadow pass of rendering, a depth map is generated from the used shadow camera. Then, during the rendering using the scene camera, now referred to as a lighting pass, every fragment is tested whether its depth value is bigger than one in the generated depth map. If so, the fragment is in shadow and this light does not illuminate it.

2.3.2 Rendering

Material node has been expanded with the possibilities to specify not only *diffuseTexture*, but also *ambientTexture*, *emissiveTexture*, *normalTexture*, *occlusionTexture*, *shininessTexture* and *specularTexture*.

Apart from the **Material** node, **PhysicalMaterial** can be used in its place. Its attributes consist of *baseColor*, *emissiveColor*, *normalScale*, *roughness*, *metallic*, *occlusionStrength*, and textures with texture mappings for all of the above. Properties *roughness* and *metallic* are the only exception to the textures, as they share the same texture file, but read different color channels.

Upon adding a **PhysicalMaterial** node to an object's appearance, the object is rendered with PBR. Otherwise, without it, Blinn-Phong lighting model is used.

This standard also touches on the topic of cast shadows. Rendering shadows is now possible with parameters newly added to light sources. Those are *shadows* and *shadowIntensity*, together with a **Shape** node's new parameter *castShadow*. Details of shadow rendering are not specified and are left up to an X3D browser's implementation.

And finally, a new light source has been added - the **EnvironmentLight** node. It defines both specular indirect light cubemap and diffuse indirect light cubemap.

2.4 glTF

In 2022, a new format became a part of the ISO standard and that would be glTF™ 2.0, a short form of "Graphics Language Transmission Format". It was standardized as a "specification for the efficient transmission and loading of 3D models" and developed by Khronos® Group. The standard describes a format represented in JavaScript Object Notation format, JSON. glTF asset in JSON might also require binary files containing geometry, animations, and other buffer-based data, and textures in image formats. glTF files are not designed to be human-readable but are developer friendly.[6]

A glTF asset file might contain more than one *scene* with its individual root nodes and its own node hierarchy with a parent-child relationship similar to VRML/X3D standard. A *mesh* can have a *material* property and such material can include various texture properties.

2.4.1 Rendering

In contrast with VRML/X3D, the glTF standard relies on PBR only, allowing the user to specify values or textures of base color, roughness property, metallic property, emissive property, occlusion, and normal property. Roughness and metallic properties are again contained within one image but in separate color channels.

Lights specification is a part of a glTF extension **KHR Lights Punctual**. An image-based lights' specification like the one in X3D 4.0 is to be found in an **EXT Lights Image Based** extension.

How exactly the scene gets rendered is up to the implementation as long as it follows the rules of mixing BRDFs as defined in this standard. In other words, if rendering speed is preferred, accurate simulation of light transport is not required.

2.4.2 Collision detection and physics

This specification and its extensions currently do not examine the problematics of physical simulations.
On the other hand the standard can contain data for other features like figure data needed for animations, and animations themselves.

Chapter 3

3D Graphics Libraries

This chapter will focus on three main open-source 3D graphics libraries: X3DOM, which is capable of presenting X3D files and including glTF2.0 files into the scene, X_ite, which can present X3D files, and Three.js, which has an official glTF loader available.

3.1 X3DOM

X3DOM is an open-source framework and runtime for 3D graphics on the Web. It is very easily integrated into a webpage by simply adding the desired version of X3DOM JavaScript file into the webpage and writing the scene directly into the HTML markup - inside <X3D> element.[7]

While X3D standards already brought a concept of **Profiles** that limit the supported **Components** and nodes described in them, X3DOM declared a new profile. This new **HTML Profile** extends the X3D-defined **Interchange Profile**. And for the purpose of an easier implementation, X3DOM does not support X3D's **Script** nodes and it is encouraged to do the desired scripting from the DOM (Document Object Model)/HTML side.

3.1.1 Rendering

Apart from supporting most of the X3D 3.3 visual features, X3DOM also adds a lot of its own functionality and support. That includes similar concepts to what can be seen in the X3D 4.0 draft.

High-resolution models can take a long time to load, but in terms of performance, once they are in the scene, FPS moves around 60 without problems.

During testing, rendering of scenes with models that use textures with transparent parts has shown significant issues, the underlying problem being that objects do not reorder correctly for rendering.

CommonSurfaceShader

As a part of the **HTML Profile** and not any X3D specification, the **CommonSurfaceShade** node can be used instead of the **Material** node, to allow multiple different-use textures. It makes concepts like normal mapping possible even without PBR, demonstrated in a following example code snippet:

```
<CommonSurfaceShader diffuseFacor='1 1 1'
        specularFactor='1 1 1' shininessFactor='1'>
    <ImageTexture containerField='diffuseTexture'
            url='diffuseMap.jpg'></ImageTexture>
    <ImageTexture containerField='specularTexture'
            url='specularMap.jpg'></ImageTexture>
    <ImageTexture containerField='shininessTexture'
            url='shininessMap.jpg'></ImageTexture>
    <ImageTexture containerField='normalTexture'
            url='normalMap.jpg'></ImageTexture>
</CommonSurfaceShader>
```

Usage of this code can be seen in the official X3DOM **CommonSurfaceShader** tutorial and example [9]. Results of this official example are show in figure 3.1.

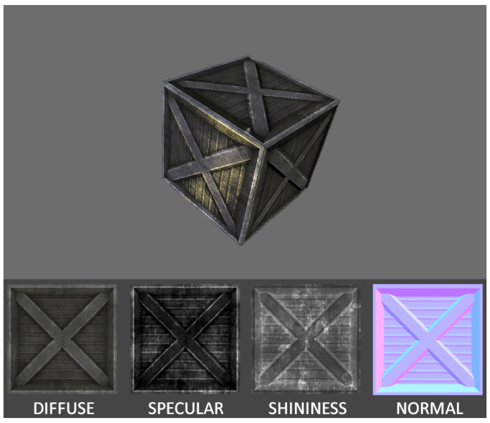


Figure 3.1: The official example of a **CommonSurfaceShader** node of X3DOM library showing diffuse, specular, shininess and normal textures applied to a model.[9]

PBR

Usage of the PBR illumination model during the rendering process can be achieved by adding a **PhysicalMaterial** node under the object's appearance, as it is specified in the 4.0 version of the X3D standard, with a few differences in the node's parameters.

X3DOM's **PhysicalMaterial** node accepts *baseColorFactor*, *metallicFactor*, etc. as the names of its attributes for PBR, whereas X3D 4.0 standard requires them named as *baseColor*, *metallic*, etc. This issue should hopefully be solved with X3DOM possible future support of the X3D 4.0 standard.

Different settings of metallic and roughness parameters can be seen in figure 3.2.

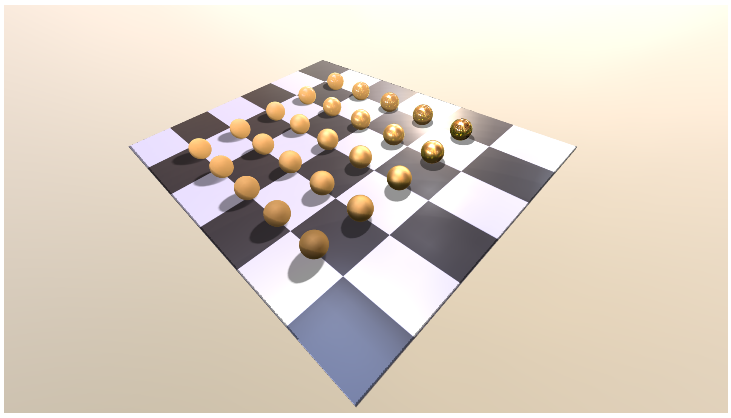


Figure 3.2: Roughness and metallic factors influence on a material in X3DOM. Object in the top corner is smooth and non-metallic, it has value of both roughness and metalness of zero, while the object on the bottom has both set to one, the maximum, which makes it fully metallic and rough. The object in the left corner is fully rough but non-metallic, and the object on the right is completely smooth and metallic.

The **PhysicalMaterial** node does accept texture maps and PBR attributes defining textures when assigned to the correct *containerField*, as shown in the following example code snippet:

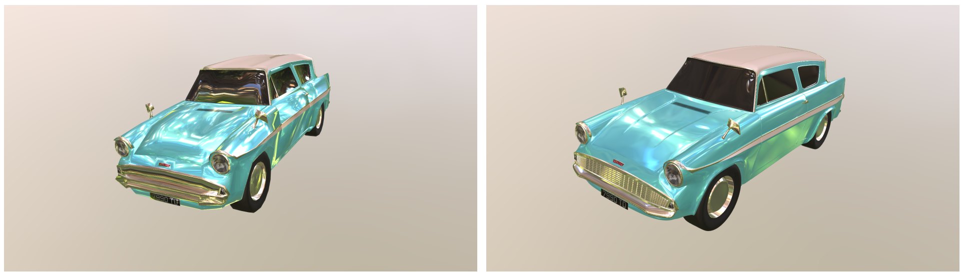
```
<PhysicalMaterial baseColorFactor="1 1 1"
        metallicFactor="1" roughnessFactor="1">
    <ImageTexture containerField='baseColorTexture'
            url='"base_diffuse.png"'></ImageTexture>
    <ImageTexture containerField='roughnessMetallicTexture'
            url='"base_metallic_roughnesss.png"'</ImageTexture>
</PhysicalMaterial>
```

Usage of this code can be seen in figure 3.3a and 3.3b. Car model used was exported from Blender software[11] and then manually adjusted to fit X3DOM's scene format.

Normal map is not included in the example, because X3DOM was not able to apply it.

It is to be noted that loading can take up to a few seconds on the low-poly model and around a minute on the high-resolution model (it has over 1.2 million vertices), plus additional few seconds for the texture loading.

A **PhysicalEnvironmentLight** node is X3DOM's version of the **EnvironmentLight** node from X3D 4.0 standard but with slight deviations. **PhysicalEnvironmentLight** has fields of an abstract light node, plus



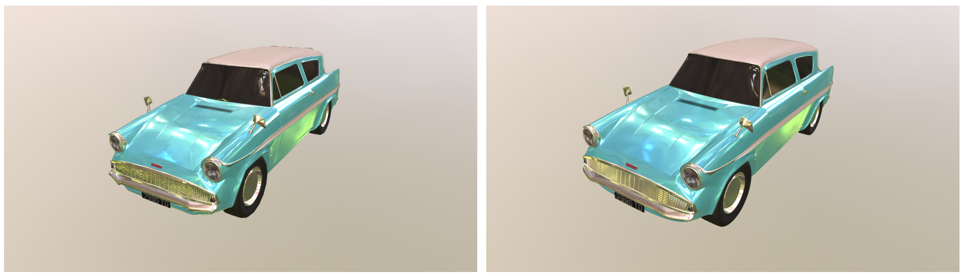
(a) : Low-resolution car X3D model. **(b) :** High-resolution car X3D model.

Figure 3.3: Usage of PBR textures shown on two levels of detail of the same X3D model with X3DOM library.

additional *diffuse* and *specular* fields, which support only an URL to a pre-computed HDR .dds file (DirectDraw Surface). X3D 4.0 specification actually did add the *diffuse* field into their specification, following X3DOM's implementation, but currently no *specular* field. Instead, the specification requires **X3DEnvironmentTextureNode** for fields *diffuseTexture* and *specularTexture*. Together with textured background, this setup is implemented in all examples created for this work. Texture sources that were chosen as default by X3DOM were used.

PBR is switched on automatically for included glTF models. Compared to X3D models, glTF models tend to be loaded into the scene a lot faster.

In figures 3.4a and 3.4b the same car model as in figures 3.3a and 3.3b can be seen, but it was exported to glTF instead of X3D and it was added to the X3D scene as a glTF model. The most noticeable change was the difference in loading speed, even the high-resolution model took at most just a few seconds. Visual quality seems also overall better.



(a) : Low-resolution car glTF model. **(b) :** High-resolution car glTF model.

Figure 3.4: Two levels of detail of the same glTF model added to a scene and shown with X3DOM.

Shadows

While X3D 4.0 introduces shadows with *castShadow* on the **Shape** node and *shadows* with settings in *shadowIntensity* in light source nodes, X3DOM's shadow casting requires only *shadowIntensity* on a light source node to be greater than zero for shadows to render. X3DOM also offers a lot of other different fields for setting desired shadow quality.

While shadows do indeed get rendered, they often create undesirable artifacts and their setting for an acceptable outcome can be quite difficult.
Loaded glTF models do not have cast shadows implemented at all. Furthermore, the inclusion of glTF models brings in rendering errors when other X3D objects have the casting of shadows turned on as shown in figure 3.5. Shadows of models loaded from X3D files can be seen through the models loaded from glTF files, while those models do not cast shadows at all.

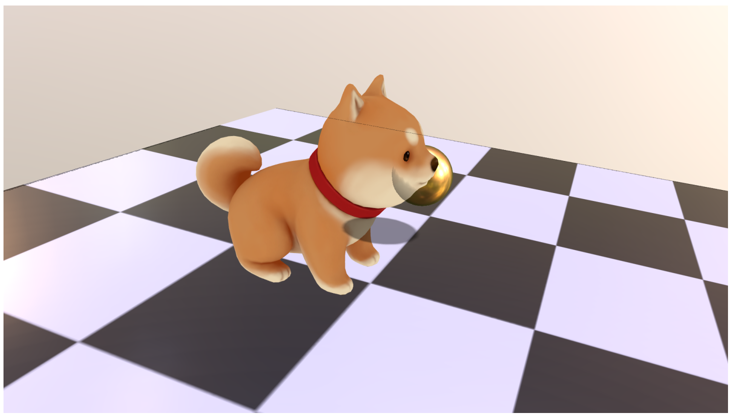


Figure 3.5: Loaded glTF model of a dog causes errors in shadow rendering. Behind the model is an X3D defined sphere above ground, casting a shadow that can be seen through the glTF dog model. The glTF model itself does not cast shadows.
Model used for demonstration is freely available, credit goes to zixisun02 [10].

3.1.2 Physics

X3DOM provides a script version with an experimental implementation of X3D's rigid body physics, but it is currently lacking and not very user-friendly. Furthermore, the **LinePickSensor** node is not included in the **HTML Profile** of X3DOM at all.

For the physics to work at all, the whole 3D scene must be a part of the original page, meaning no objects that shall be affected by physical simulation can be added through an **Inline** node. This causes the page source to be disorienting and messy.

There is no user documentation of this component on the X3DOM website other than that of the individual nodes, but a paper on its development was published and it describes the principles and concepts behind the implementation as well as a few performance tests.[12]
It also reveals that the implemented physics run on ammo.js which is a direct port of Bullet Physics Engine into JavaScript.

Concepts

Every rendered **Shape** node that shall be affected by the physical simulation must be parented by a **Transform** node. This node will be controlled by the simulation and will move the **Shape** node accordingly.

For every such **Shape**, a **CollidableShape** outside of the transformation hierarchy should exist. A **CollidableShape** must contain a **Transform** node in a *containerField* of value *physics*, and a **Shape** node that best describes the object's physical shape for the physical simulation. This **Transform** node must be connected with the rendered **Shape** node's **Transform** parent through USE/DEF statement.

The **CollidableShape** node is then inserted with the help of USE/DEF statements into a *containerField* of value *geometry* in a **RigidBody** node. The **RigidBody** node resides inside of a **RigidBodyCollection** node outside of the transformation hierarchy, in a *containerField* of value *bodies*. Slightly different from X3D's standard, if the **RigidBody** node has a *mass* of value zero, it acts as a static object not affected by physical forces.

Finally, the **CollidableShape** node should be included in a **CollisionCollection** node with USE/DEF statement through the *containerField* of value *collidables*. Such **CollisionCollection** resides in a *collider containerField* of a **CollisionSensor** node.

With this setup, physical simulation can be tested. Rendered **Shape** node is set up truly only for rendering, while the **CollidableShape** node sets the geometry for collision detection and links rendered **Shape** node's transform into the simulation. The **RigidBody** node sets the needed attributes for the physical simulation itself.

Testing

Very simple scenes containing only translated built-in shapes work fairly well. The initial *translation* in the **Transform** parenting node and in the **CollidableShape** node must be of the same value.

Problems do arise when the desired initial transformation includes *rotation* to a non-default value. That is the case for most X3D files exported from Blender software, because of different coordinate axes' orientation. The **CollidableShape** indeed rotates and functions as it should, but the rendered result is rotated with a nonsensical value.

Another big issue is not taking complex geometries under the **CollidableShape** into account, even those with just tilted top faces. Instead, X3DOM probably chooses an axis-aligned bounding box to work with, even for convex models, as shown in figure 3.6, testing a non-novex model's behavior can be seen in figure 3.7, which also did not give satisfactory results.

Physical constraints do work to some extent. X3DOM does have a few examples working with simple objects on their page, but with the limited

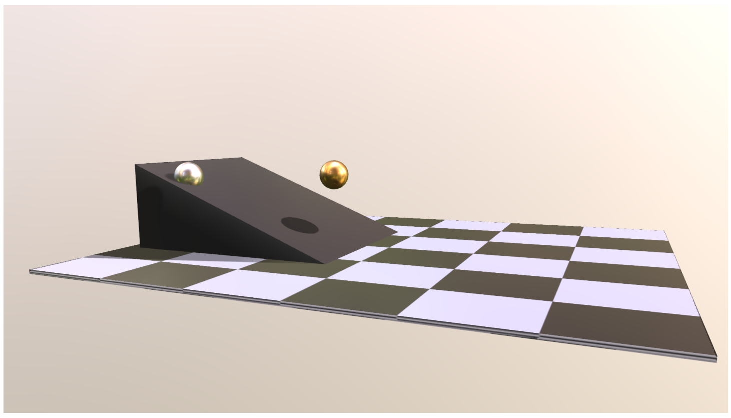


Figure 3.6: Spheres should have collided with the tilted platform, but instead both collided with an invisible block, resulting in the left sphere being partly in the tilted platform and the right one levitating on top of it.

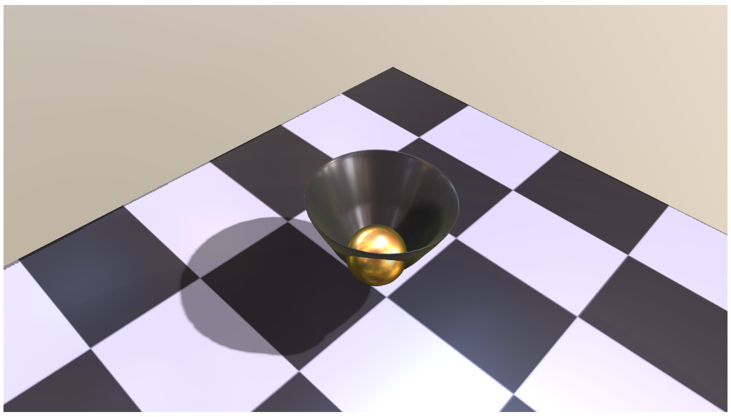


Figure 3.7: Collision detection between a sphere and a non-convex object in X3DOM. The colliding geometry of the bowl shown was not created correctly and let the sphere pass through its walls.

knowledge of how exactly the scene should be set up, a simple pendulum scene demonstrated in other implementations was not achieved in X3DOM for this work, as can be seen in figure 3.8.

3.1.3 Communication with External Software

Scripting and direct modification of the 3D scene are made very easy with X3DOM because the 3D scene is directly pasted in the HTML tree inside the <X3D> tag. Addition and deletion of nodes is detected and the scene gets rerendered automatically, although issues were encountered with trying to set fields directly after node addition.

Redirection of messages about collisions and user interaction is possible.

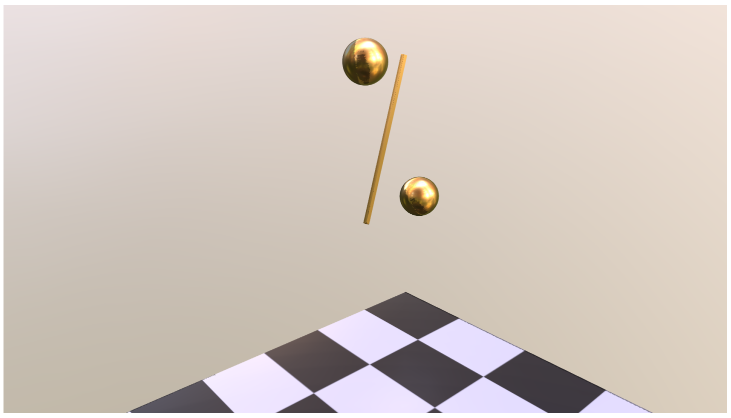


Figure 3.8: Broken pendulum created with usage of **SingleAxisHingeJoint** node. Satisfactory result was not achieved. Although the bottom sphere is swinging correctly, the pendulum imitation as a whole breaks.

3.1.4 Licence

X3DOM is dual-licensed under MIT and GPL licenses.

MIT license, otherwise known as Expat license, is a permissive free software license giving users the freedom to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the software licensed under it as long as the license text included in it is included also in the software used in the final product.

A dual license allows users to choose to follow the rules of one of the two licenses included. For commercial use, in this case for MATLAB and Simulink, the MIT license is more suitable, as GPL, General Public License, would require sharing the whole project freely to the public.

3.2 X_ite

X_ITE is a 3D JavaScript library which uses WebGL for 3D rendering.[13] Similarly to X3DOM, no plugin installation is needed, the user shall just include X_ite's script in their page and position the element <x3d-canvas>.

X_ite currently fully supports X3D's **Immersive Profile**, as well as 94% of X3D specified nodes.

3.2.1 Rendering

Like X3DOM, X_ite also implements most visual features of X3D 3.3 standard, with few new features from the X3D 4.0 draft. A significant drop in performance even with more complex scenes has not been noticed.

Considering that X_ite follows the standard far more strictly than X3DOM, features like **CommonSurfaceShader**, **RenderedTexture** or other exclusive parts of X3DOM's **HTML Profile** are not supported. Though the user is free to program their own custom shader.

PBR

While X_ite does support turning on the PBR illumination model for objects with a specified **PhysicalMaterial** node under **Appearance**, the desired effect cannot currently be achieved, because the implementation of the **EnvironmentLight** node is missing.

Thus, X_ite does not reach the same goals in this matter as X3DOM does. This difference with X3DOM can be seen together with different settings of metallic and roughness parameters in figure 3.9.

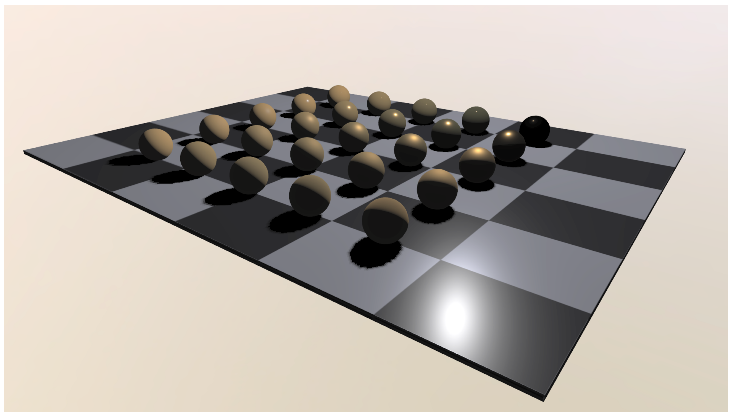


Figure 3.9: Roughness and metallic values influence on a material in X_ite. The scene is the same as shown in figure 3.2. Note that **EnvironmentLight** node is not yet implemented in X_ite.

Hopefully, this crucial node's functionality will be added with the release of the standardized X3D version 4.0.

While there is no prepared shader for material textures when it comes to Blinn-Phong lighting model, adding PBR material textures under the **PhysicalMaterial** node into a correct *containerField* does result in them getting applied, as can be seen in figure 3.10.

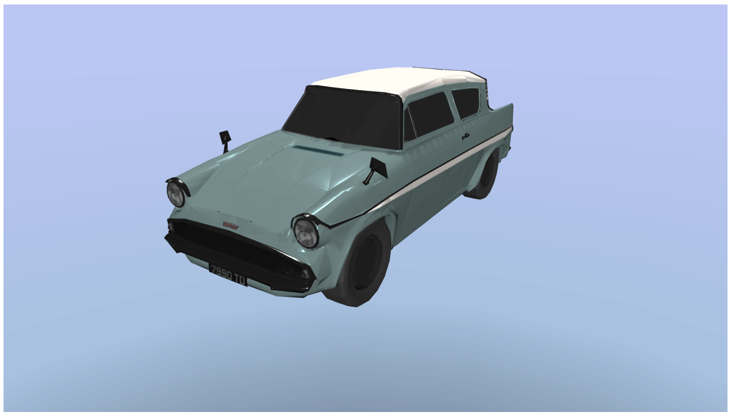


Figure 3.10: Usage of PBR textures shown on an X3D model with X_ite library. Note that the environment mapping and image based lighting is not yet implemented.

Shadows

X_ite turns on the functionality for the shadow rendering when light nodes have a field *shadows* set to true. After that, the only influence user has on shadows' rendering is *shadowIntensity* field on the light node and *castShadow* field on individual shape nodes, which is set to true by default.

3.2.2 Physics

Concepts

Although the use of **RigidBodyComponent** is not documented in X_ite either, its use seems easier compared to X3DOM.

The user does not need to make a **Shape** node inside a **Transform** node that would get linked to a **CollidableShape** node. Creating a **Collidable-Shape** node by itself is enough for both rendering and physical simulation. Its reference should then be given into a *collidables* field of the **Collision-Collection** node, as well as *geometry* field of the **RigidBody** node through USE/DEF statements and *containerField* fields.

Unlike X3DOM, X_ite has a **CollidableOffset** node implemented and functional, and it is used to reposition the collision geometry within a **Rigid-Body** node.

Testing

Basic shape collisions work without many problems. That can be seen in many examples on the X_ite site.

User-defined shapes also work in collision simulation. However, a collision between two non-basic shapes is not detected unless one of them is static. However, this is not unusual for physics engines. Figure 3.11 shows such working example.

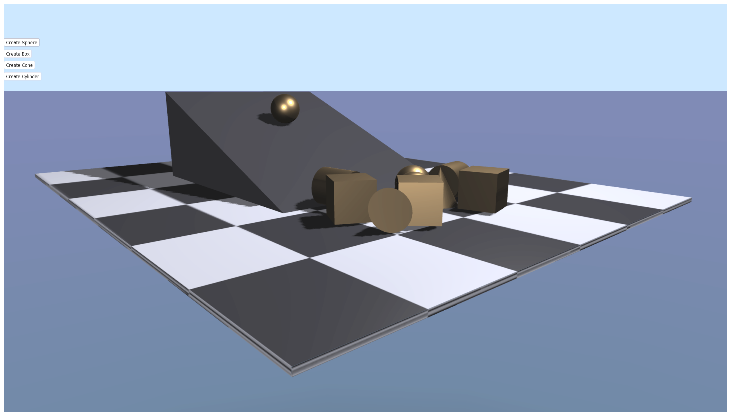
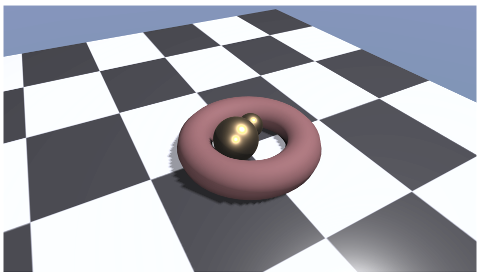
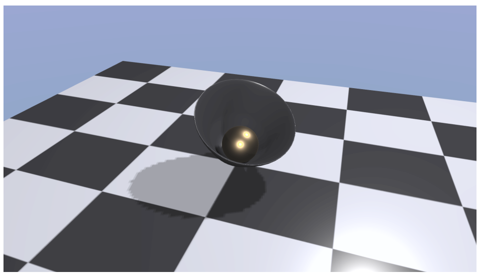


Figure 3.11: Static tilted platform collisions with basic shapes in X_ite.

Same applies to non-convex shapes. X_ite does not create a convex hull collision shape for them, but simulates collision with a non-convex shape as shown in figures 3.12a and 3.12b.



(a) : Torus model collision with X3D spheres.



(b) : Bowl model collision with an X3D sphere, same scene as in figure 3.7.

Figure 3.12: Example of non-convex shape collision with basic shapes in X_ite.

Physical constraints are also working. Example in figure 3.13 is a demonstration made possible with following code snippet:

```
<SingleAxisHingeJoint
            anchorPoint='0 6 0'
            axis='0 0 1'>
    <RigidBody USE='top' containerField='body1'/>
    <RigidBody USE='bottom' containerField='body2'/>
</SingleAxisHingeJoint>
```

SingleAxisHingeJoint limits the movement of the two given rigid bodies to only a rotation around a specified axis centered on an anchor point. In this example, **CollidableOffset** was used for the bottom sphere, so that it is under the same **RigidBody** node as the stick shape in the middle. It stays on the bottom of it, even though it is a basic sphere shape with a local coordinate system originating at its center.

Drop in FPS can be noticed in around 100 basic shape objects colliding in a scene. For all three tested browsers, Mozilla Firefox, Google Chrome, and Opera, FPS dropped to 30-40.

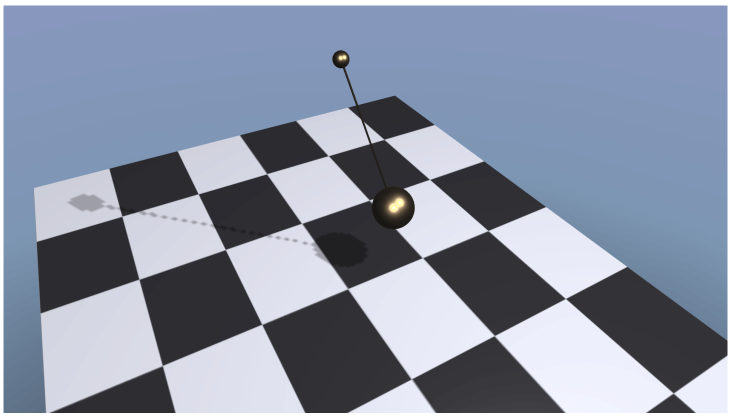


Figure 3.13: Swinging pendulum realized with the **SingleAxisHingeJoint** node in X_ite, fully functional.

3.2.3 Communication with External Software

X_ite takes a different approach than X3DOM when it comes to the modification of the scene from an outside source. It is done through X3D standard's **Script** nodes and their user-defined **field** nodes. They can be read from inside the script as values or even call functions from the scene in the script. The supported language of **Script** nodes in X_ite is ECMAScript.

X_ite offers predefined methods for modification of the X3D scene through the script and predefined classes corresponding to X3D field types standard.

3.2.4 Licence

X_ite is published under the GNU General Public License v3.0, which makes it free software in terms of usage, modifications, or redistribution, and adding it into a project makes the whole project also licensed under the same terms. That makes it unsuitable for MATLAB and Simulink's commercial use.

3.3 Three.js

Three.js is a JavaScript 3D graphics library using WebGL [15]. It takes a very different approach to the building of a scene compared to X3DOM or X_ite. Through JavaScript alone, the user defines a camera, renderer, lights, shapes, and animation loop to get a simple scene image.

The Three.js script itself does not include crucial things like viewpoint movement control or loading background textures, but its repository of examples, loaders, utility functions, and additional classes is fairly big and mostly documented [16].

3.3.1 Rendering

Due to countless contributors to the library, rendering options and features are very vast. This chapter will limit itself to PBR illumination of imported glTF models and will compare features that can be found to some extent in X3DOM/X_ite.

Performance-wise, during the testing of visual features, excluding the time during the loading of models, FPS stayed consistent at 60.

PBR

Mesh in Three.js is defined by geometry and material. Geometry can be a Three.js basic shape like a cube, sphere, cone, etc., or a loaded model mesh. The main material types are **MeshLambertMaterial** or **MeshPhongMaterial** which loosely correspond to X3D's **Material**, and **MeshStandardMaterial** or **MeshPhysicalMaterial** which are used in PBR. **MeshPhysicalMaterial** is an extension of **MeshStandardMaterial** and its use in rendering precedes X3DOM's current features. For example, it supports *clearcoat* property or *transparency* that is physically based with a customizable index of refraction, IOR. Different settings of metallic and roughness parameters can be seen in figure 3.14.

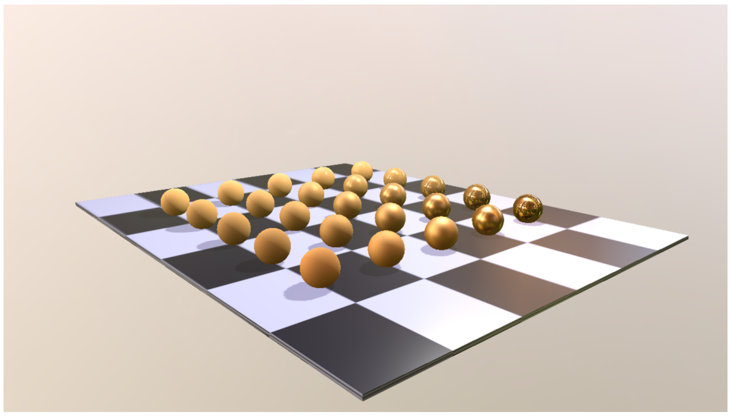
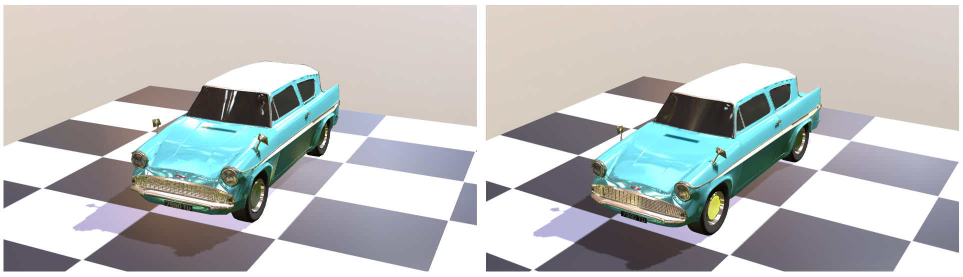


Figure 3.14: Roughness and metallic values influence on a material in Three.js. The scene is the same as shown in figure 3.2, but exported to glTF instead of X3D.

A glTF loader is included in Three.js example loaders. It allows the user to add glTF models to the scene. Models created for this work were exported from Blender and needed to be corrected manually in order to be rendered without problems.

Upon load, their material characteristics are automatically assigned and the illumination model used for their rendering is switched to PBR. A similar setting to environment light's effect defined in X3D 4.0 can be achieved by assigning environment texture to Three.js scene's *environment* property.



(a) : Low-resolution car glTF model. **(b) :** High-resolution car glTF model.

Figure 3.15: Two levels of detail of the same glTF model added to a scene and shown with Three.js.

Shadows

Turning on rendering of shadows cast by objects is achieved by setting the properties *castShadow* and *receiveShadow* on desired objects to true.

Also, loaded glTF models exported from Blender need to have their material edited. The property *doubleSided* must be set to false and *side* of *material* must be described as **FrontSide**.

Then shadows are set up and working, contributing to similar or better-looking results compared to X3DOM/X_ite.

3.3.2 Physics

Concepts

For this work, ammo.js JavaScript library, a direct port of Bullet Physics Engine into JavaScript, is used for testing physics. It is the same one X3DOM uses.

Its integration into Three.js starts by specifying rigid bodies with mass, motion state, and local inertia for every desired object. Colliding shape set is then added to a defined physical world. The simulation changes transformations of those colliding shapes with discrete simulation steps the user must set up and control. For the rendered result to also move, the user has to apply the transformation change from simulation to rendered shapes.

Testing

Ammo.js allows the user to create not only basic collision shapes but also convex hulls or even full collision non-convex meshes. While defining basic collision mesh corresponding to Three.js counterparts is straightforward, correct geometry processing needs to be implemented by the user to be able to pass it to a convex hull collision shape constructor.

Loaded glTF models need to be processed in such way if the user does not wish to approximate them by basic shapes. This work uses a method [18] which reconstructs a triangle mesh of single vertices of the original geometry into ammo.js-defined structures. Then ammo.js constructor is used to create a convex hull collision mesh. Finally, correct world and local transforms are applied to a rigid body.
The basic physics test is shown in figure 3.16.

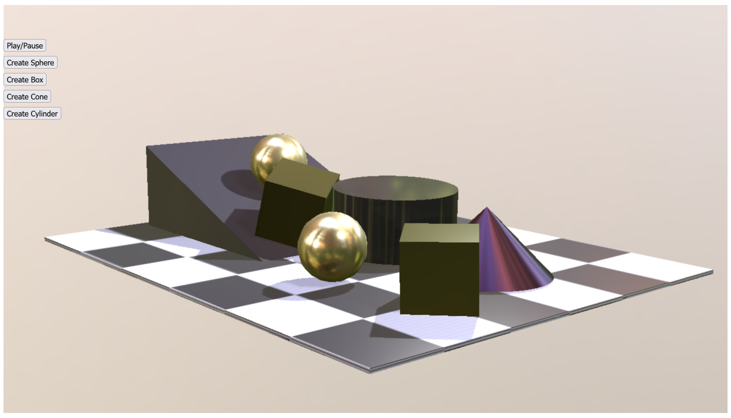


Figure 3.16: Static tilted platform collision with basic shapes in Three.js.

Compared to X_ite, a significant drop in FPS can be seen at around 1000 colliding basic shape objects, down to 20-30 FPS.

Due to an easy export of glTF from Blender compared to X3D, even more complex collisions could be tested. The tests are shown in figures 3.17 and 3.18.

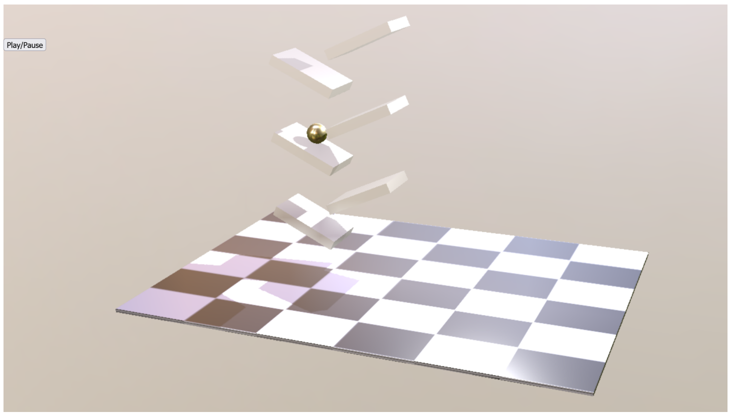


Figure 3.17: Multiple static tilted platforms collide with basic sphere in Three.js.

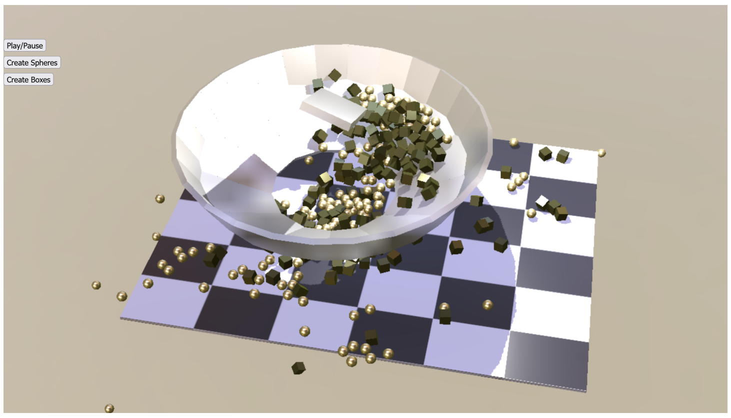


Figure 3.18: Bowl created out of many tilted platforms collides with hundreds of basic shapes in Three.js.

Ammo.js also supports physical constraints. A pendulum example in figure 3.19 demonstrates once again the use of a hinge constraint. Apart from the features that X3D specifies, ammo.js is also able to simulate cloth and soft body to some extent.

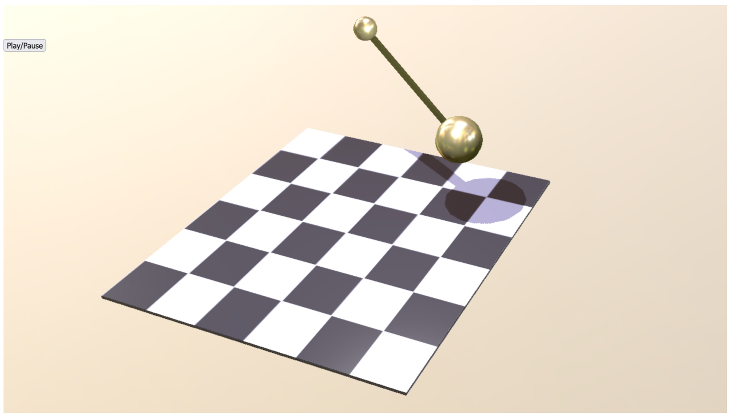


Figure 3.19: Swinging pendulum imitation with **btHingeConstraint** class in Three.js.

3.3.3 Communication with External Software

Considering the whole scene is created, set and run from JavaScript code, communication with external software is without any limitations other than those of JavaScript and the external software itself.

3.3.4 Licence

Three.js is licensed under the MIT license, therefore same restrictions as X3DOM's MIT part of its dual license are applied.
Ammo.js has its own license which allows free use, redistribution, modification, and commercial use without the need to acknowledge the use of it in the final product. If the library gets modified, it shall be specified as such and not pose as the original distribution. Given license has to be added to all redistributions.

Chapter 4

Simulink 3D Animation

This chapter will overview the current implementation state of the tool Simulink 3D Animation.

MATLAB is a computing environment as well as a programming language developed by The MathWorks, Inc. It is widely used together with Simulink, a block diagram environment used to design systems with multidomain models, simulate before moving to hardware, and deploy without writing code.[19]

Simulink® 3D Animation™ is a tool under MATLAB software that links Simulink models and MATLAB algorithms to 3D graphics objects in virtual reality scenes.[20] With this tool, the user is able to load a 3D scene from a VRML or X3D version 3.3 file, supporting the **Immersive Profile** of the X3D version 3.3 specification.

4.1 Usage of Simulink 3D Animation

To open a scene from file of for example *vrbounce.x3d* in a viewer or edit its content in an editor, the user should use one of the following commands:

```
% open a viewer
vrview('vrbounce.x3d');
% open an editor
vredit('vrbounce.x3d');
```

For direct modification of the scene from code, the scene's reference can be obtained through:

```
% get currently opened worlds' references
w = vrwho;
```

Through the scene's reference, named nodes and their fields can be then accessed. In the following example, the file needs to describe a **Transform** node called *Ball* containing a sphere shape with **Material** node of *Ball_material* for the following code to work properly.

```
% set ball's diffuse color to yellow
w.Ball_material.diffuseColor = [1 1 0];
% create a cone object in the scene
coneShape = vrnode(w.Ball, 'children', 'NewShape', 'Shape');
coneGeometry = vrnode(coneShape, 'geometry', 'Cone', 'Cone');
% delete ball
delete(w.Ball)
```

To work with a 3D scene in Simulink, a **VR Sink** block can be used to write data into the scene and a block **VR Source** to read data from the scene. Under the block's parameters, desired scene file can be selected together with named nodes' fields that the Simulink model should have access to. Upon running the model, the scene gets updated on changes in those fields in a desired sample time, which is also adjustable in the block's parameters.

4.2 Current Implementation

The software can be divided into 4 main implementation parts: MATLAB interface, internal scene, canvases and editor/viewer as can be seen in the simplified software architecture outline in figure 4.1. It is important to note, that the current implementation maintains two main branches - main version, which is based on Java and OpenGL, and an experimental version which uses JavaScript for rendering. This work will be focusing on enhancing the experimental version.

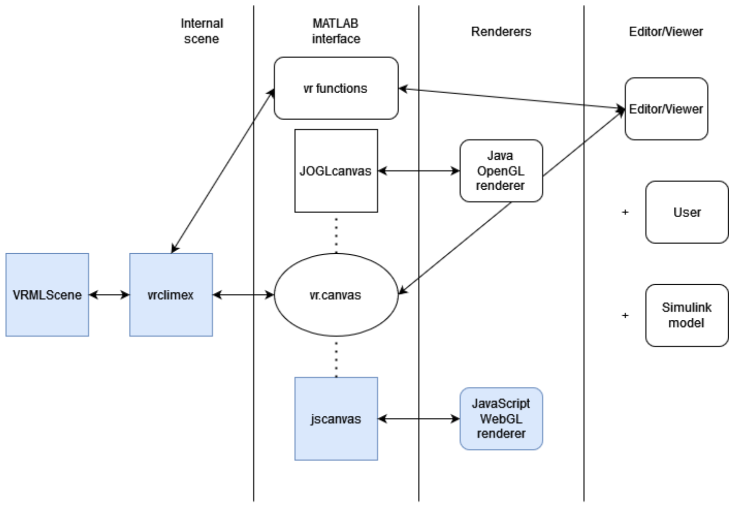


Figure 4.1: Simplified outline of Simulink 3D Animation implementation. Blue components indicate changes done for the purposes of this work.

4.2.1 MATLAB Interface

Simulink 3D Animation comes with a collection of MATLAB functions mainly for scene opening, visualization, modification and closing. These are written mostly in MATLAB, but they use underlying C++ code for any data exchange with the scene.

In MATLAB, C++ programs can be called through MEX functions, MEX standing for MATLAB executable, which behave like built-in MATLAB functions. Simulink 3D Animation works with MATLAB's C matrix API functions rather than the now recommended modern C++ matrix API.[21] In Simulink 3D Animation, MEX functions are used for communication with the internal scene, through **vrclimex** which serves as a main module of MEX interface for Simulink 3D Animation.

Using MATLAB functions from C++ program is also possible, through **mexCallMATLAB** function, which is able to call MATLAB's built-in functions or even user-defined functions based on their name, pass parameters to them and get return values, both in the form of a pointer to a **mxArray** object that MATLAB works with.

4.2.2 Internal Scene

All opened scenes are handled by the C++ portion of the software and their states are kept and updated there through vrclimex regardless of the chosen renderer.

Upon load of the scene file, the scene gets parsed into an internal representation of a **VRMLScene** class loosely based on OpenVRML Library. The implementation follows VRML and X3D specification in terms of an inheritance hierarchy, but being originally built for VRML only, the inclusion of X3D nodes is done through a **DynamicX3DNode**, while all supported VRML nodes are represented by their own classes.

DynamicX3DNode is derived from a **VRMLNodeProto** class, which represents implementations of VRML prototypes, here defined under a class **VRMLNodeType**. **DynamicX3DNode** instantiated in a scene either encapsulates a VRML node instance, having its own implementation added in type definition or a mixture of both. A difference between a **Shape** node containing a **Geometry** node loaded under the VRML specification versus one loaded under the X3D specification can be seen in figure 4.2.

Every scene has an assigned id, which is used during communication with the MATLAB interface. For similar reasons, every loaded valid node of the scene holds an attribute of *name* - internal name, which is either defined in the scene file or generated to be unique during the scene load.

For the purpose of updating renderers, events and scene state, an internal timer is started upon any successful scene opening. This timer repeatedly calls **drawnow** function in vrclimex, which manages these tasks. **Drawnow** function can also be called from MATLAB, with a function called **vrdrawnow** there, and it is usually very frequently called by a Simulink model to update the rendered frame.

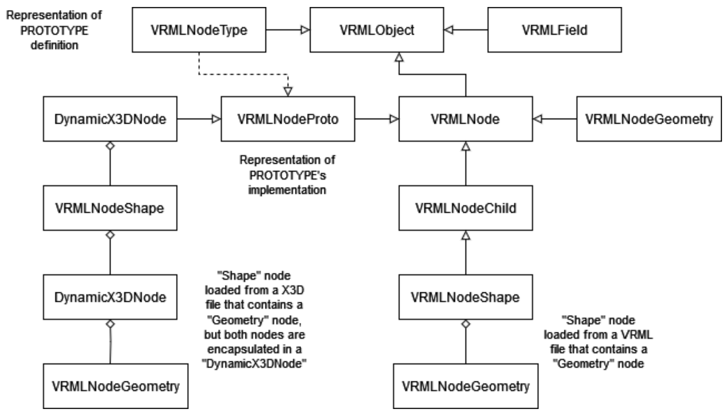

Representation of PROTOTYPE definition
VRMLNodeType
VRMLObject
VRMLField
DynamicX3DNode
VRMLNodeProto
VRMLNode
VRMLNodeGeometry
Representation of PROTOTYPE's implementation
VRMLNodeShape
VRMLNodeChild
DynamicX3DNode
"Shape" node loaded from a X3D file that contains a "Geometry" node, but both nodes are encapsulated in a "DynamicX3DNode"
VRMLNodeShape
"Shape" node loaded from a VRML file that contains a "Geometry" node
VRMLNodeGeometry
VRMLNodeGeometry

Figure 4.2: Internal scene class hierarchy design shown on an example.

4.2.3 Canvases

Java based canvas

To render scene view into MATLAB figure, **vr.canvas**, otherwise called "virtual reality canvas", is used. Each canvas comes with a set of its *canvas properties*, which influence the rendered result - for example whether to render the geometry as wireframe or not or which viewpoint to choose from. These properties do not influence the state of the internal scene, but their initial values are set from the internal scene as well as from the renderer.

The virtual reality canvas uses a renderer written in OpenGL and encapsulates rendered frames in MATLAB figures through Java. For the purposes of rendering with OpenGL, all geometry in the scene can be triangulated and represented as **IndexedFaceSet** of triangles with re-computed normals, texture coordinates, indexing, and vertex colors.

Experimental canvas

Experimental canvas, referred to as **jscanvas**, is used in a MATLAB **uifigure**, which does not heavily rely on Java compared to the figure object used with the main canvas. The experimental canvas contains an **HTML UI component** capable of displaying HTML5 and JavaScript content.

Internally, the HTML UI component uses Chromium browser and to allow communication with MATLAB, offers a communication channel through a **Data** variable provided by MATLAB on both sides. Callbacks can be used to react to changes in the variable.

The conversion between MATLAB values and JavaScript objects is done by MATLAB through encoding into and decoding from JSON on both sides. Although communication with the component through websockets could be technically possible, it is closed off for security reasons. Similarly, the loading of resources, media or source code is intentionally very limited. Everything has to be saved locally in the directory where the main HTML source file is opened from and other resources loaded at runtime have to be of specific allowed file format.

The experimental renderer takes a completely different approach to rendering the displayed frame. The HTML UI component used in the experimental virtual reality canvas runs a modified version of X3DOM 1.8.2. Upon canvas creation, the internal scene produces a string of valid X3D scene description which is sent over to HTML UI component and added into the HTML markup to be registered by X3DOM and fully loaded there. Thus from that moment on, the HTML UI component renders the scene independently on the internal scene and has to receive information on scene modifications and send back information on user interaction.

In its current state, the capabilities of the experimental canvas are very limited. All the limitations that X3DOM has apply here. Communication protocol between the experimental canvas and HTML UI component is not implemented at all, which results in mainly two issues.

Firstly, the scene is frequently not being loaded, because its representation is sent to the component sooner than the scripts in it are loaded.

And secondly, upon changes in the internal scene or requests to the canvas, the canvas does not wait for the resulting change in canvas properties from the HTML UI component and instead offers out-of-date values.

4.2.4 Modification of the scene and canvas properties

Most common modifications of the scene come in forms of node addition, node deletion and node field value setting. Those calls are executed in the internal scene immediately, but the visual propagation of the changes is not and it differs between the main and experimental version of the tool, together with the setting of canvas properties.

Java based Simulink 3D Animation

In the main version of Simulink 3D Animation software, **drawnow** call redraws the scene if any visible changes were made and it propagates a new frame to all canvases connected to it.

Values of canvas properties are held within a **ViewerOpenGL** class, they are updated from within the scene and from the virtual reality canvas to be always up-to-date. Retrieving the value in the MATLAB interface through the virtual reality canvas calls into the C++ code and returns the wanted value directly from **ViewerOpenGL**.

The scene modification propagation to the viewport can be seen in figure 4.3.

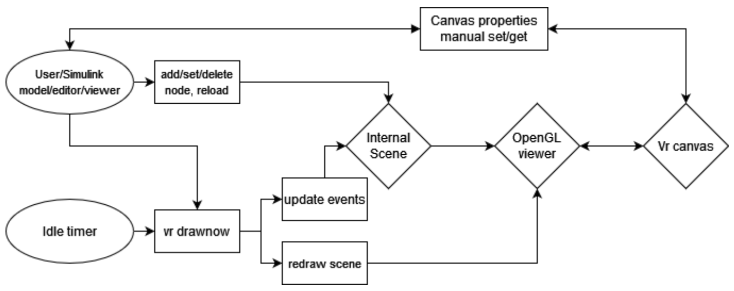

Canvas properties manual set/get
User/Simulink model/editor/viewer
add/set/delete node, reload
Internal Scene
OpenGL viewer
Vr canvas
update events
Idle timer
vr drawnow
redraw scene

Figure 4.3: Modification of scene and canvas properties propagation in Java based Simulink 3D Animation.

There is no asynchronicity in the run of the software, thus there has never been any need for a communication protocol.

Experimental Simulink 3D Animation

As stated above, the experimental version of Simulink 3D Animation renders the scene separately once loaded, because it does not have direct access to the internal scene representation and has to be simultaneously updated on all visible changes through the HTML UI component communication channel.

Upon scene modification, the internal scene gets immediately updated as in the Java-based version, but the information on the changes gets at the same time pushed into queues for the renderer to receive later on. The update messages come in a form of a string, where keywords are separated by a hash symbol. An example of such message string can be seen down below:

```
1#MESSAGE_ID_ADD_NODE#N0000021153D2DFB0#
<Sphere id='Sphere1' name='N0000021153D2C5B0' DEF='N0000021153D2C5B0' >
</Sphere>#
```

The number at the beginning is in code referred to as a message ID but is never assigned a number other than 1. It is ignored on the HTML UI component's side during parsing. The next keyword specifies the incoming change and it is followed by parameters specific to that change. For node addition that would be a name of a desired parent node and then a description of the node that is to be added, already printed out in X3D format, prepared to be inserted directly into the HTML markup.

The **drawnow** function does not have any effect on the propagation of the changes to the renderer, instead, a second timer was added directly into the virtual reality canvas implementation, which makes the canvas repeatedly check the internal scene for updates in the update queues in hard-coded intervals. Upon fetching them, they get sent through the HTML UI component communication channel where they are parsed and processed.

Because the HTML UI component runs separately from the MATLAB code, issues arise.

The main one that comes with the approach explained above is the independence on Simulink's or user's **vrdrawnow** calls, which could serve as the way to ensure similar behavior to synchronous execution of the code. In a situation, where a currently used virtual reality canvas gets deleted and information about the new current viewpoint is required immediately from the canvas, it does give out an outdated value even if **vrdrawnow**, which is supposed to guarantee up-to-date values, is called before the information request.

Furthermore, Simulink calls **vrdrawnow** after every simulation step, but because it has no effect on the message retrieval, several calls for gradual visual changes get frequently sent together with only the last one having any effect.

The scene modification propagation to the viewport in experimental version can be seen in figure 4.4.

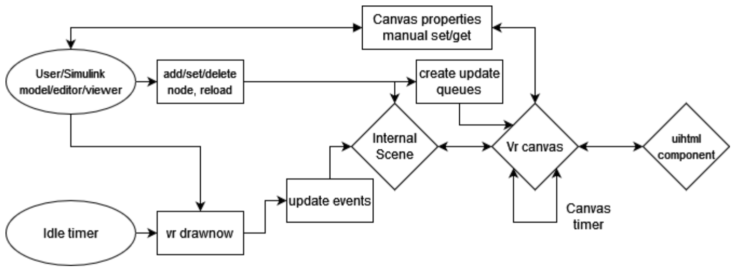


Figure 4.4: Modification of scene and canvas properties propagation in experimental Simulink 3D Animation. Note the existence of the second timer.

Chapter 5

Suggested Modifications and Implementation

Based on the review of JavaScript renderers, communication capabilities of MATLAB with HTML UI component and current implementation of the Simulink 3D Animation tool, changes in the supported nodes, communication protocol and experimental renderer are suggested, and implemented. This chapter will go over the most important decisions, reasoning behind them and the will describe the principles of their implementation.

5.1 Supported nodes

In order to massively enhance the visual capabilities of the tool, defining new node types for the internal scene representation was suggested. Those nodes include mainly those of the 4.0 version of X3D specification - **PhysicalMaterial** and **EnvironmentLight**. Furthermore, enhancing the current definitions of **Material**, **Shape** and all light nodes with new fields would allow the use of PBR features, advanced texturing of models and shadow casting if implemented in the used renderer.

5.1.1 Implementation

Before new node definitions and enhancement of current node definitions of new fields could take place, version control had to be solved. Because Simulink 3D Animation supports scene export after modification, newly added nodes and fields require a flag of they fall under the X3D 4.0 version, so they do not get saved upon export, because the resulting file would not be a valid one under the X3D 3.3 version of the specification. In the future, export of X3D 4.0 content will be desired, but while the specification draft is not yet approved, the software does not support this feature yet.

Example of node enhancement source code - enhancing **Material** node with a *specularTexture* field and setting its 4.0 version flag, can be seen on the following source code snippet:

```
// specular texture
MaterialType->addExposedField("specularTexture",
                               VrmlField::SFNODE,
                               VrmlSFNode(),
                               "X3DSingleTextureNode");
MaterialType->setFieldVersion("specularTexture",
                               X3DVersion::X3D_40);
```

Because the software aims to support the **Immersive profile** of the X3D specification, all node definitions and fields were updated to version 4.0 as specified for the profile in this version with the exception of the **EnvironmentLight** node, which still has an unstable definition and is left out in official XML schemes for X3D file parsing. To get similar results, the **EnvironmentLight** node type was still added to the software, but for easier implementation and use, the definition deviates from the current draft of X3D 4.0 standard by having *topUrl*, *bottomUrl*, *frontUrl*, *backUrl*, *leftUrl* and *rightUrl* fields for cube map texture of the environment similarly to **Background** node fields. The XML schemes did need to be manually changed accordingly.

After these modifications, the software is now capable of loading nodes and fields required for the **Immersive profile** of X3D 4.0 specification into the internal scene.

5.2 Communication protocol

The current implementation of the experimental canvas frequently does not load the scene at all, misses the first incoming scene modification calls, and does not offer up-to-date virtual reality canvas properties. The string format of the messages currently used also heavily complicates the possibility of screen capture or any binary data transmission.

The suggested approach was to implement a new communication protocol, between the virtual reality canvas and the code running in the HTML UI component, with a message confirmation system, which would ensure that all requested properties are not outdated. Such protocol could also use the native MATLAB objects as the content to be sent because the HTML UI component's communication channel is able to automatically convert them into JavaScript objects and vice versa, and thus there would be no need for manual string parsing. The removal of the second timer operating in the experimental canvas and using the internal one through the **drawnow** calls instead could also enhance the performance and fix the synchronization issues explained earlier.

5.2.1 Software architecture changes

For the removal of the second timer, changes needed to be made in the concept of connecting the experimental canvas to the internal scene. When an experimental virtual reality canvas gets created, it registers into a chosen scene for updates. These get stored in update queues and upon a **drawnow** call, an update message is produced from all of them and that is sent back to MATLAB canvas through a MEX callback function.

The scene modification propagation to the viewport in the new experimental version can be seen in figure 5.1.

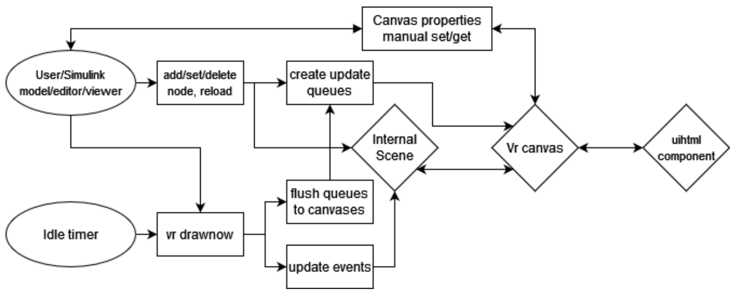


Figure 5.1: Modification of scene and canvas properties propagation in experimental Simulink 3D Animation, newly implemented.

5.2.2 Communication of canvas with HTML UI component

Because the communication channel the HTML UI component offers is not well documented from the technical standpoint, tests needed to be done to ensure that this channel is reliable and does deliver all inserted messages in the order that they were sent. This functionality was confirmed.

MATLAB natively does not support a general **wait** function, because MATLAB without specific extensions is not multi-threaded. The experimental virtual reality canvas does need to block execution during html and script loading, during scene loading, upon requests to the renderer from the canvas, e.g. requesting a screen capture, and upon synchronization of canvas properties.

MATLAB does, however, offer a pair of **uiwait** and **uiresume** functions, which were originally designed for blocking the execution until a certain input is received from the user, for example, the user closes a modal window or clicks a button on it. For this reason, they have to be always associated with an uifigure, that, when closed, would unblock the execution as with the modal window example.

The **uiwait** function blocks the execution, but in the meantime, callbacks, like the one that can be added to the HTML UI component's communication channel, stay fully functional, and thus when a condition is met during the processing of a callback, **uiresume** function can be called and execution resumes where it was stopped.

With these functions, waiting for HTML and script loading to finish is implemented through a confirmation, then the scene representation gets sent over to the component and canvas again waits for confirmation of scene loading.

To ensure the delivery of up-to-date canvas properties, a system for message confirmation was implemented as well.

Upon every **drawnow** call in the scene, when update queues get flushed into the canvas, even if they're empty, a **refresh** method is called on the canvas. This method sets the *WaitedForID* property, which starts at a value of zero upon canvas creation and 1 is gradually added to it for every sent message during **refresh** or from other canvas methods.

With every received message, the renderer running in the HTML UI component confirms the message with the highest ID number it received and processed the message it belongs to. While the messages get processed by the code, dirty flags are set upon modifying anything that canvas properties depend on and those changes are sent back to the canvas together with confirmation.

When a property is required from the canvas and the property depends on scene modification updates, using **vrdrawnow** beforehand is recommended, because it sends over any yet unsent changes from the update queues together with a higher message ID than is currently confirmed. When, in this state, the canvas is requested to provide a value of such dependent property, it instead waits on the condition of:

```
while (obj.WaitedForID > obj.ConfirmedID || ~obj.SceneReady)
    uiwait(obj.mfigure);
end
```

5.2.3 Data format

As was mentioned earlier, the suggested format of data being sent to the HTML UI component from the canvas was a MATLAB object and JavaScript object the other way around. The HTML UI component's communication channel's implementation ensures the conversion of these data representations.

Every valid message from canvas to HTML UI component's code is a MATLAB object with properties *type*, *data* and *messageID*. There are currently 18 implemented types of messages, most notable being *refresh*, then those setting canvas properties, e.g. *headlight* or *wireframe*, those controlling viewpoint with the options like **gotoNextViewpoint** or **gotoDefaultViewpoint**, and then there is also a message of type *capture* for screen capture request. The *data* property differs from type to type.

For reasons explained in detail later in this work, all portions of **refresh** messages that describe a scene or part of the scene are encoded into JSON format directly in the C++ portion of the code using a library called RapidJSON.

Every valid message from the HTML UI component renderer back to canvas in MATLAB is a JSON object with properties *type* and *data*. The main three types of messages are *confirm*, *canvasEvent* and *response*. *Data* property is again dependent on the type of the message.

Confirm type of message contains together with data that include updates for renderer-dependent canvas properties, sensor updates and also the highest processed message's ID.

CanvasEvent informs the canvas of the loading status of the HTML page, source codes and then the scene itself.

Response message is currently used for the transmission of screen capture data.

5.3 Renderer

While the renderer currently used, X3DOM 1.8.2, offers many features, its main advantage compared to X_ite is its license, and compared to Three.js is its ability to directly work with X3D file format. Other than that, both of these two other libraries surpass its capabilities significantly. There has not been a new stable release of X3DOM since June 13th of 2021 with little over 50 contributors on its github repository, while Three.js gets updated almost daily, has over 1700 contributors, is widely used and has an active user forum.

To add to that, in a long-term perspective, because the goal of the work on the experimental version of the software is for it to have all the important features of the main version and more, many significant problems can already be expected if X3DOM stays as the used renderer, mainly the support of **LinePickSensor** for simplified collisions, which X3DOM does not implement at all, while also not having any user-friendly methods for ray casting into the scene. Another big issue would be the use of a stereo camera, which is also not supported in X3DOM.

Being a library designed to display contents of an X3D scene, all features and modifications needed by Simulink 3D Animation had to be done directly in the library's source code, which is not very well documented, and combining singular features of the library to achieve the desired capability can pose as a fairly difficult task.

The suggested modification for the rendering component of the experimental version of the software was a transition to the use of the Three.js library. Although there are significant differences between the approach to scene representation and interaction between Three.js and the X3D specification, it still offers much more features and an easier way of implementing needed ones, because of its imperative nature.

This transition required export of the scene description from the internal scene into the HTML UI component renderer, in a form that would be convertible to Three.js objects and principles, together with the implementation of loading of the content, management of scene modifications and implementation of user interaction.

Taking the vast amount of features that need to be implemented for a full transition into consideration, this work focused mainly on the implementation of core mechanisms for scene loading, scene modifications and support of rendering features explored in chapter 3.

5.3.1 Scene export

Because the internal scene representation is stored in a module that does not have access to the C Matrix API of MATLAB and Three.js is not able to load the X3D printout string that the scene is currently able to produce, a different approach had to be taken. Considering that JavaScript natively works with the JSON format, upon performance, usability and license consideration of libraries JSONCpp, nlohmann/json and RapidJSON, RapidJSON was chosen and used, mainly because of its performance and its SAX style API.

The term "SAX" originated from Simple API for XML. It is borrowed for RapidJSON's JSON parsing and generation.[22] While most JSON libraries for C++ require building a document with objects, which can be then stringified, SAX API is event-based and the objects are created based on a sequence of **Key** and **Value** events generated by a **Writer** object, which can be passed between scene nodes as a pointer during scene traversal.

Upon experimental virtual reality canvas creation, the internal scene is traversed and a method called **printThree** is executed on every scene node. A **VRMLNode** class is the base class of all scene nodes and thus class inheritance could be used for, in principle fairly simple, implementation of scene export into the desired format. Because not all node types that the internal scene is able to load are supported or need to be loaded into the new renderer, the **VRMLNode** class **printThree** method writes a null value through the given writer and all derived classes which should give out an output are expected to override this method.

The starting point of the scene traversal for scene export can be seen in the following source code snippet, the first node processed is a **Group** node serving as the scene's root.

Part of source code of **VRMLScene** class method producing loadable scene represenation for JavaScript code can be seen down below:

```
using namespace rapidjson;
StringBuffer jsbuffer;
Writer<StringBuffer> jswriter(jsbuffer);
jswriter.StartObject();
// clear temporary flags for all nodes,
// we'll use them for marking USEd nodes
d_nodes.clearFlags();
// scene graph description
jswriter.Key("scene");
d_nodes.printThree(&jswriter);
jswriter.EndObject();
```

Before the contents of any node get processed, a node type is noted into the new representation of the node and then a flag is checked, whether the node has been processed already, indicating the DEF/USE relationship on the node. If the node has indeed been processed already, a **Key** of *use* is added with a **Value** corresponding to an ID of the node, that would otherwise get added to the representation together with all supported fields representation.

The ID currently used for the nodes takes inspiration from the X3D string export for X3DOM, being the text representation of the node's pointer's address. Field representing classes also implement a **printThree** method according to the values they are able to store and are usually executed from a **printFieldThree** method defined in the **VRMLNode** class, which checks whether the field exists and is of non-default value. If these conditions are passed, it sets **Key** to their name and calls their **printThree** methods to produce **Value**. An example of a node **printThree** method can be seen in the following source code:

```
void VrmlNodeShape::printThree
    (rapidjson::Writer<rapidjson::StringBuffer> *writer)
{
  writer->StartObject();
  if (this->printIdentifiersThree(writer, true))
  {
    this->printFieldThree(writer, "appearance");
    this->printFieldThree(writer, "geometry");
    this->printFieldThree(writer, "castShadow");
  }
  writer->EndObject();
}
```

Because the Java-based version of the software processed and triangulated all inserted geometry for use by OpenGL, methods covering this feature could be reused to avoid the need to process the geometry in the JavaScript renderer in the exact same way, which would otherwise be needed because apart from **BoxGeometry** and **SphereGeometry**, the built-in Three.js geometries are not flexible enough to fit the X3D specification, and the base **BufferGeometry** expects triangulated input only.

The following example shows how the red ball object used to demonstrate the representation of the objects in formats of VRML and X3D in chapter 2 is represented as a JSON object, if the file describing it gets loaded in the internal scene and exported for the new renderer (line breaks and spacing was added for readibility):

```
{"node":"Transform",
 "name":"000001997F0DB020",
 "internalName":"Ball",
 "children":[
 {"node":"Shape",
  "name":"000001997F0DDDD0",
  "internalName":",0000000000000008",
  "appearance":
  {"node":"Appearance",
   "name":"000001997F0DC340",
   "internalName":",0000000000000009",
   "material":
   {"node":"Material",
    "name":"000001997F0DD880",
    "internalName":"Ball_material",
    "diffuseColor":[1.0,0.0,0.0]}},
  "geometry":
  {"node":"Sphere",
   "name":"000001997F0DEA90",
   "internalName":",000000000000000A"}}],
 "scale":[1.5,1.5,1.5],
 "translation":[0.0,20.0,0.0]}
```

Processing of prototype nodes definitions is also not needed, because it is already done during the internal scene creation, so during the scene traversal, **VRMLNodeProto** classes are encountered and they encapsulate nodes that implement them, which can be traversed and processed as any other scene node. It is important to note that every node loaded from an X3D file is encapsulated in the **DynamicX3DNode**. This class does not have any effects on the exported scene representation by itself directly, it only passes the process onto all the nodes it encapsulates.

Scene modification

The **drawnow** call for the internal scene produces an object in JSON format to be sent to the renderer through the canvas as well. While the scene is being modified through the MATLAB interface or on its own by running

Script nodes, information on those changes is saved into the update queues. Currently, there are three of them, one for node addition, deletion, and field synchronization. Other updates sent to the renderer include also reload requests and current scene time information.

In the JSON update object, an array of all changes is created for each update queue.

The representation of node addition is produced in the same way as it would during the initial scene export process, but only the added node's representation is prepared, together with an ID of the parent node. The contents of the node do not get processed during the creation of the update object but in the moment of the node addition to the scene itself. It is done so in order to avoid producing duplicates if another node is added to this new node and both are included in the update queue at the same time. RapidJSON does support the insertion of *RawValue* as a **Value**, so this kind of pre-processing does not pose any issues.

Needed information on field synchronization includes the ID of the modified node, the field's name and a new value, which is printed in the same way it would during the initial scene export through the **printThree** method as well.

Deletion of nodes produces only an ID of the affected node.

However, there is an issue that arises with the fact that the export mechanism uses pre-processed geometry by the internal scene.

When a field value is changed, a node is deleted or added and such modification affects any node representing geometry, it has to be recalculated and resend to the renderer. Hovewer, with the DEF/USE mechanism, the information on where all the affected nodes are in the scene is completely missing.

Such changes require additional scene traversal of modified branches to locate those nodes, process their geometry again and send this information over to renderers.

5.3.2 Scene import

As the call for the build of the scene from provided JSON representation is noticed, not only is the scene graph that Three.js uses for rendering being produced but so is a map of all loaded nodes. This map accepts node IDs as key values and stores reference to all instantiated nodes with the same ID in an array under the the key.

In contrast with the X3D principle of DEF/USE mechanism, Three.js does not provide a way to merely reference an object and process its existence only during rendering. All nodes that are directly used for rendering need to be individual objects added to the scene graph, and as a result, the map of nodes holds references to all instances under every node ID.

Scene class

A scene class encapsulates the scene object used by Three.js for rendering, while also managing scene navigation, user interaction, rendering settings propagation to the scene nodes and holding the map of node IDs.

It also provides a **buildFcnMap**, a map of functions for node building and updating out of the JSON representation, with node types as keys.

Node classes

Similarly to the internal scene class inheritance hierarchy, the JavaScript renderer's code now defines a class for every supported scene node. Each node always implements a constructor, **clone** method possibly with **copy** method, **init** method, **set** method and **delete** method, while also including a reference to the **scene** object, a property that indicates its ID, internal name and node type. Names of the node classes always start with "X3D" and these classes were all newly implemented for this work.

The class constructor always takes in the scene instance reference and optionally additional ones specific to the individual node type. It usually also sets default values to all properties representing supported fields.

Nodes that are added to the scene graph for Three.js are represented by derived classes from those provided by Three.js that represent the same feature. Other nodes are either represented by new base classes or derivations of those.

Upon instantiation of a node through the constructor, the **init** method should be called, which takes the JSON representation of itself as the parameter. Among additional steps specific to individual node types, the **init** function always includes setting the ID, calling the **set** function with the JSON object passed to it from the argument, and registration into the scene node map.

The **set** method is used for both node building in node addition and in field values synchronization. It checks the passed JSON representation for supported properties as its keys and upon retrieving their values updates its state accordingly. When a new node is built, the JSON representation passed to this method sets all the non-default properties, and then with every field value synchronization sets only those affected.

For cases when the node is marked as used, the **clone** method is provided. It takes in the same arguments as the constructor. If the node has to be instanced again to be added to the scene graph, this method calls the constructor, uses a defined **copy** method that copies all the properties, and children nodes in some cases, and returns a new reference. In case when the node does not need to be represented in such way, the method returns the old reference and internally stores a reference to the node that contains it and needs to be updated on changes in the added node. Reference to the node added this way does not get reinserted into the map of nodes in the scene object as it does in those re-instantiated during the execution of the **copy** method.

The **delete** method is called either on a deletion call of a given node or from its parent node, which could originate in scene reloading as well, by calling this method on a root node.

It is responsible for disposing of all used GPU-related resources allocated by its instantiation, executing **delete** method on all its children nodes or nodes in its properties, removing references to itself from its parents or removing itself from the scene graph and unregistering itself from the scene node map.

For nodes that do not get instantiated anew during cloning, they need to check whether their list of parents is not empty upon the removal of a parent every time. If it is, it calls the **delete** method on itself, disposing of itself accordingly.

Shape Node

The biggest difference between the X3D standard principles and the Three.js approach to the representation of the scene comes in the form of the **Shape** nodes. While the X3D **Shape** node can exist independently on its *geometry* and *appearance*, Three.js provides different scene objects for different geometry types they are able to present - all polygonal mesh can be presented with **Mesh** objects, but lines should be presented with **LineSegments** objects and points with **Points** objects.

Those native object classes can be assigned with a suitable geometry and a class-specific material, while the X3D standard specifies a **Shape** node capable of holding any kind of geometry node and **Appearance**, which also is not dependent on the type of geometry assigned.

This means, that the **Shape** node cannot be represented with a native scene object unless it is given a geometry, or rather the type of geometry it will be using. Changing the geometry type also implies a needed change of the object using it, as in disposing of the old one and instantiating a new one that is able to present the geometry correctly.

The **Appearance**, if defined, needs to be applied to the newly created object as well, being converted to values that the new material suitable for the new geometry can use.

For the reasons stated above, it is necessary for the renderer to remember values specified by the internal scene export rather than those converted for and used by Three.js.

X3DShape class

A direct representation of the X3D **Shape** node is implemented through a new **X3DShape** class, which is derived from the Three.js class Object3D. It is able to be added to the scene graph and be influenced by the transformation hierarchy.

This class holds properties of, among others, *geometry* and *appearance*. And these two together directly influence the state of a *_shape* property, which, if the geometry is defined and valid, holds the real object defined for Three.js to render, which is added to the scene graph under the instance of the **X3DShape**.

Through this class, changes in both its geometry and its appearance are propagated to the real object, reacting to their addition, modification and deletion accordingly.

Canvas property setting has also an effect on this class, as global shadows setting, conversion to PBR setting, and wireframe setting all get propagated through it into the *_shape* object if present.

Geometry classes

The naive approach would be to just instantiate a new geometry object for every new **Shape** node that has one defined, but with the DEF/USE mechanism of X3D standard, geometries can be re-used once defined.
Every instance of the newly implemented geometry classes keeps track of **X3DShapes** that hold it, through a list. When **X3DShape** gets cloned, it only acquired a reference to this node and thus the geometry is shared between all the **X3DShape** using it.
Upon addition of the **X3DShape** node to the list, a new *_shape*, which class is dependent on the geometry type, is produced and inserted under the **X3DShape** and it calls for updates of newly instantiated Three.js material it is using if appearance and appearance's material are defined. When geometry gets modified or deleted, needed changes get propagated into all dependent *_shapes* as well.

There are three important methods implemented to further solve the differences between the Three.js approach and X3D specification: **update-MaterialColors**, **updateMaterialSide** and **UpdateMaterialAll**, which executes both previous methods.
Because Three.js material implementation required the indication of whether to use vertex colors information specified in the geometry, this property needs to be set on all *_shapes*' materials from the geometry.
Similarly, a setting of Three.js material's *side* property is dependent on the X3D value of *ccw* field - whether the geometry is defined counter clock-wise and value of *solid* field - whether the surface of the geometry should be

one-sided or double-sided. Combinations of those two fields result in a setting of the real material of *_shape* to be back-sided, front-sided, or double-sided.

Appearance

The X3D **Appearance** node is represented by a class **X3DAppearance** which can hold references to classes **X3DMaterial**, **X3DPhysicalMaterial**, **X3DTextureTransform** and **X3DTexture**, while it itself keeps track of all **X3DShapes** that contain it. It also provides a method **getRealMaterials** to retrieve all references to real materials of *_shapes*. It is used by material and texture classes to propagate their changes.

Both material classes keep track of all appearances that use them as well as a list of real materials in the scene that they currently influence. This list gets updated through the previously mentioned **getRealMaterials** method. The material itself then automatically propagates individual changes to this list right on any modifications of its properties through setters. An example of a simple property setter can be seen in the following code snippet:

```
// setting the emissiveColor field on a material
set emissiveColor(val)
{
  this._emissiveColor = val;
  for (const mat of this._materials)
    if (mat.emissive)
      mat.emissive.setRGB(...val);
}
```

The red ball object previously shown in a VRML, X3D and exported JSON representation is a part of a *vrbounce* official example of Simulink 3D Animation. This example, loaded into the internal scene, exported and then imported into the new renderer can be seen in figure 5.2.

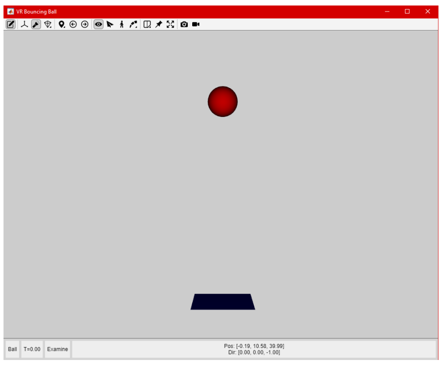


Figure 5.2: Simulink 3D Animation *vrbounce* official example showing import of the scene into the new renderer.

Three.js texture objects cannot be directly shared, because their mapping onto the meshes is influenced by a **TextureTransform** node, but the underlying image data can be shared and are shared between all materials that use it.

The **X3DTexture** class keeps track not only of all the appearances and materials that use it but also of how they are used specifically, whether that is a normal map, diffuse texture, and so on. When the texture is first created, it loads the actual Three.js texture into its *_texture* property, searches for all appearances using it, even if it is through a material, clones the *_texture* for every single appearance, calls for **TextureTransform** node application on the cloned texture, requests all real materials on the appearances and updates those with the cloned and transformed texture. This process allows for texture used under different appearances to have different transformations set, while at the same time sharing the texture between all real materials dependent on one appearance. And all of this does not make a deep copy of the underlying image data.

A source code example for such update can be seen down below, it implements base/diffuse texture addition to all required real materials:

```
// go through all registered x3d materials
// with base or diffuse texture
for (const x3dmat of array)
{
  const newmap = this._texture.clone();
  // go through all appearances that use them
  for (const app of x3dmat._x3dappearances)
  {
    // apply appearance's texture transform
    app.setTextureTransform(newmap);
    const materials = app.getRealMaterials();
    // and apply it to all real materials
    // linked to that appearance's shape
    for (const mat of materials)
    {
      if (mat.map) // remove old texture
        mat.map.dispose();
      mat.map = newmap.clone();
      mat.needsUpdate = true;
    }
  }
  newmap.dispose();
}
```

Because texture loading is handled as an asynchronous task in Three.js, the process explained above does not happen until the texture is fully loaded, but directly after the loading is finished.

X3DLinePickSensor

The **LinePickSensor** node of X3D specification was implemented through Three.js's **Raycaster** class. On every simulation step, **update** method is called on all enabled sensors in the scene. The line pick sensors thus activate, and prepare collision data to be sent back to MATLAB and Simulink with together with the message confirmation.

Because the line geometry specified for the **LinePickSensor** does not have to come in a form of individual straight lines for every sensor, but instead as a general line geometry with many line segments, every segment is processed individually. Source code for processing of an individual segment can be seen below:

```
// apply parent transform to ray start and end points
this._parent.localToWorld(startPoint);
this._parent.localToWorld(endPoint);

// get segment length
const length = startPoint.distanceTo(endPoint);

// get segment direction
const direction = endPoint.clone().sub(startPoint).normalize();

// set raycaster
this._scene._raycaster.set(startPoint, direction);
this._scene._raycaster.far = length;

// get all intersecting objects
const intersects =
    this._scene._raycaster.intersectObjects(...this._pickTarget);
```

Mapping of X3D nodes to Three.js classes

X3D Node	Three.js class	relationship	status
Networking Component			
Anchor	Group	inheritance	implemented
Inline	Group	inheritance	implemented
Grouping Component			
Group	Group	inheritance	implemented
Switch	Group	inheritance	implemented
Transform	Group	inheritance	implemented
Rendering Component			
IndexedFaceSet	BufferGeometry	inheritance	implemented
IndexedLineSet	BufferGeometry	inheritance	implemented
IndexedPointSet	BufferGeometry	inheritance	not supported yet
Shape Component			
Material	MeshPhongMaterial	encapsulation	implemented
Material	LineBasicMaterial	encapsulation	implemented
Material	PointsMaterial	encapsulation	not supported yet
PhysicalMaterial	MeshStandardMaterial	encapsulation	implemented
PhysicalMaterial	LineBasicMaterial	encapsulation	implemented
PhysicalMaterial	PointsMaterial	encapsulation	not supported yet
Shape	Mesh	encapsulation	implemented
Shape	LineSegments	encapsulation	implemented
Shape	Points	encapsulation	not supported yet
Geometry3D Component			
Box	BoxGeometry	inheritance	implemented
Sphere	SphereGeometry	inheritance	implemented
Text Component			
Text	BufferGeometry	inheritance	not supported yet
Font	BMFont	inheritance	not supported yet
Lighting Component			
DirectionalLight	DirectionalLight	inheritance	implemented
SpotLight	SpotLight	inheritance	implemented
PointLight	PointLight	inheritance	implemented
Texturing Component			
ImageTexture	Texture	encapsulation	implemented
PixelTexture	Texture	encapsulation	not supported yet
MovieTexture	VideoTexture	encapsulation	not supported yet
Navigation Component			
Billboard	Group	inheritance	not supported yet
LOD	LOD	inheritance	not supported yet
Viewpoint	Object3D	inheritance	implemented
Navigation Component			
Background	Mesh	encapsulation	implemented

Table 5.1: Table presenting the mapping how X3D nodes are implemented using Three.js. Other supported nodes do not derive from or encapsulate any objects of Three.js

5.3.3 Navigation

Navigation in the scene was one of the minor reasons for transitioning the renderer from the modified X3DOM renderer to a Three.js-based one. Both the main and experimental versions of Simulink 3D Animation implemented the navigation in a fairly outdated way for today's standards.

The navigation principles that were requested include: the camera can orbit around a selected target in the scene, the camera can rotate around the center of its local coordinate system, the camera can zoom in and out on a selected target in the scene proportionally based on a distance to the target, the camera movement can be controlled by keyboard input and the camera is grounded under the WALK navigation type of X3D specification.

While X3DOM does not provide any documented method of ray-casting into the scene for retrieving the target point used for the navigation, Three.js does, thus implementation of the navigation system with the latter library was easier.

A viewpoint binding mechanism was also implemented directly according to the X3D specification.

5.3.4 User Interaction

Although the current state does not support any **Sensor** nodes yet, basic user interaction can be done through the editor, where the user can select nodes by simply clicking on them.

This functionality was implemented through Three.js provided ray-casting abilities, upon clicking on a valid visible mesh object, the *internalName* of a the object is read and sent back to MATLAB as an event for node selection. Selection of nodes is then visualized through Three.js's **OutLinePass** post-processing feature.

Another implemented interaction feature comes with the **Anchor** node implementation in **X3DAnchor** class, which bounds the camera to a specified viewpoint upon clicking on its child nodes.

Chapter 6

Results

This chapter presents the results of the work on implementation explained in the previous chapter. Most of the examples used for testing, comparisons and showcase are the official examples of Simulink 3D Animation.

6.1 Communication Protocol

With no adjustments to the official examples, the experimental viewer before proposed and implemented changes was not guaranteed to offer up-to-date information on its virtual reality canvas properties. It was also very prone to stop functioning as a result of scene modification information being passed to it, while the scene has not been loaded yet. Furthermore, being independent on **drawnow** calls from the scene meant that modifications of multiple simulation steps were applied at the same time, resulting in the loss of visible changes.

The newly implemented communication protocol does guarantee waiting on page load, scene load and confirmation of messages when a renderer-dependent property value is requested.

Due to the object nature of the new protocol, rather than the string format used before, screen capture image data transfer from the renderer to the canvas was made possible and implemented as well.

Communication stress tests, which failed before the implementation with a very high frequency, all passed with this new implementation. Results can be seen in the table 6.1.

Testing of the correct opening of multiple canvases failed immediately as the scene representation was not even received by more than one opened canvas before the implementation.

Test on the correctness of the viewpoint index was executed by repetition of creating an experimental canvas, accessing the canvas property for viewpoint information and closing the canvas.

Repeated node addition and its followed deletion caused MATLAB to crash before the new protocol every time the canvas timer called for a scene update message in the moment of node addition being registered and having to be printed out, but the node itself was actually deleted in the internal scene.

Testing on repeated object translation, moving an object on a circular path before the new protocol implementation proved that the update messages did get sent in groups of steps that should have been individual because of a used **vrdrawnow**.

Testing for	Number of rounds	Failed before	Failed now
Creating new canvas	10	10	1
Viewpoint index after scene load	50	50	0
Deleting and creating an object	100	100	0
Setting translation of an object	2000	1633	0

Table 6.1: Results of three stress test types performed on the experimental version of Simulink 3D Animation showing a comparison between the number of failed rounds before new implementation and after.

6.2 Scene export and import

Current scene export and import for the experimental renderer supports most of the nodes the internal scene by itself does with the exceptions of a **Text** node, a **Font** node, points nodes, fields specific to ambient light, a **Billboard** node, a **MovieTexture** node, a **PixelTexture** node, sound nodes and sensor nodes, which have not been implemented yet.

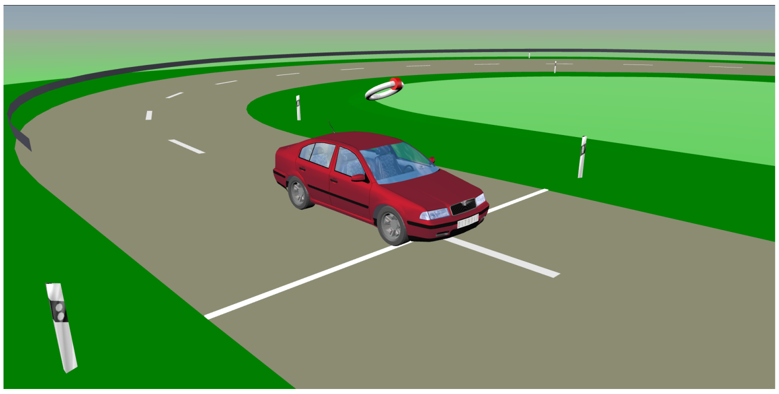
On the other hand, for example, nodes **TriangleStripSet**, **IndexedTriangleFanSet** and **TriangleFanSet** are not supported by X3DOM's **HTML Profile** and thus were not rendered even with modified X3DOM viewer, but can be loaded into the Three.js based viewer and rendered now.

Most of the scenes used by Simulink 3D Animation official examples are able to fully load with minor visual differences, as can be seen in figure 6.1a as it was rendered before the implementation of proposed modifications in comparison with the figure 6.1b, which is produced by the new renderer.

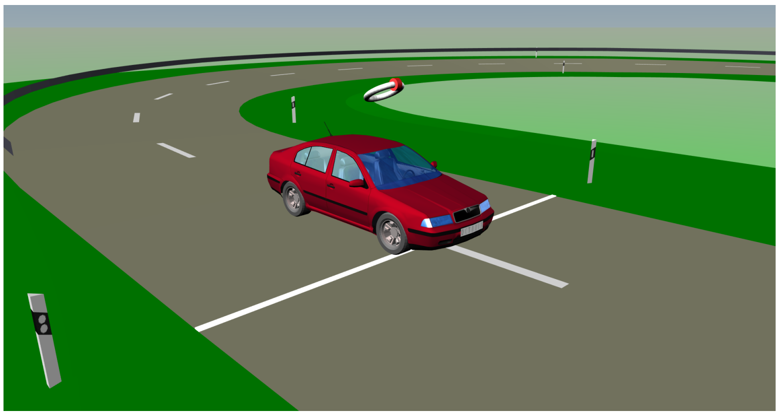
Loading times of the scenes vary, the official examples taking at maximum of seconds, but scenes heavy on detail with millions of vertices do not get loaded in reasonable time, similar to how the modified X3DOM version performed, or even the main version based on Java in some cases does.

6.3 X3D 4.0 standard support

Upon modification and enhancement of the internal scene's supported node types and implementing support for given scene nodes representation in the Three.js-based renderer, visual features explored in chapter 3 are now possible to use.



(a) : Scene rendered with modified X3DOM, before implementation.



(b) : Scene rendered with Three.js based renderer, after implementation.

Figure 6.1: Scene from the official example of *vr_octavia* rendered with experimental viewer before and after the implementation.

Added support of **PhysicalMaterial** node and modified **EnvironmentLight** node allows the creation of 3D scenes that get rendered using PBR. The previously used car model loaded in the new renderer is shown in figure 6.2.

Experimental virtual reality canvas also has a newly implemented functionality to force PBR rendering even on old scenes, converting **MeshPhongMaterial** used in Three.js very roughly into **MeshStandardMaterial** for all meshes with this material type as well as adding example background and environment map to the scene.

Figures 6.3a and 6.3b demonstrate on an official Simulink 3D Animation example referred to as *vrmaglev* this functionality, as well as newly implemented support for casting of shadows, which work well with **PointLight** and **SpotLight** but cause rendering and performance issues with **DirectionalLight** because Three.js does not natively support shadow cascades.

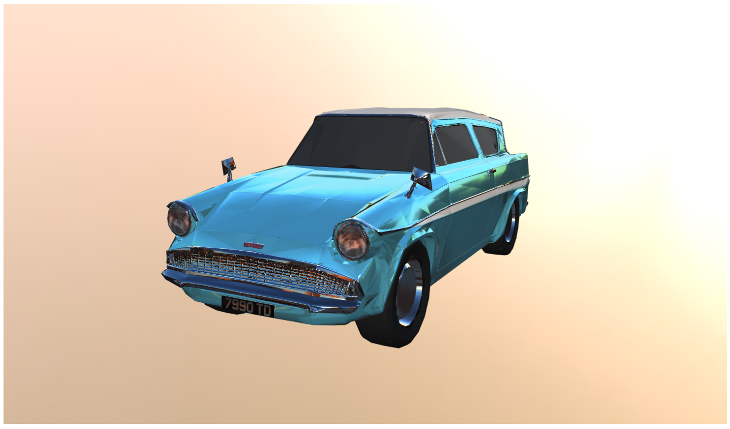
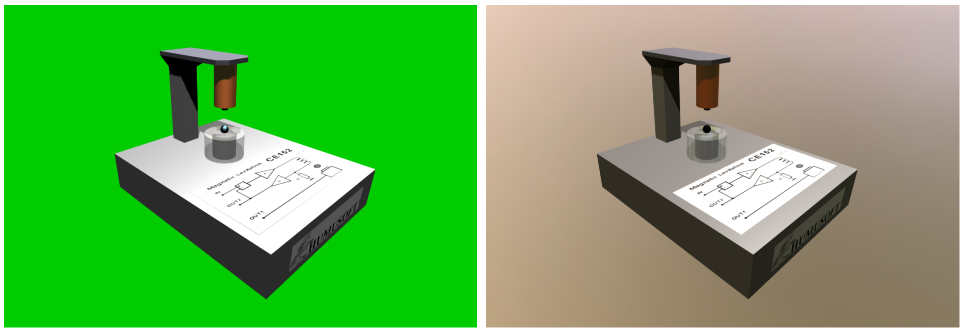


Figure 6.2: Car low-resolution X3D model with PBR textures shown in the new Simulink 3D Animation viewer.



(a) : Example scene with original materials and background.

(b) : Example scene with enforced PBR, example texture background and environment.

Figure 6.3: Simulink 3D Animation *vrmaglev* official example shown in the new viewer.

Classic **Material** node now supports the use of advanced texturing with very similar results to the use X3DOM's **CommonSurfaceShade**, with the exception of *shininessTexture*.

6.4 Collision detection

LinePickSensor node functionality needed for Simulink 3D Animation official examples of *vrmaze* and *vrcollisions_lidar* is fully implemented in the experimental version as can be seen in figures 6.4 and 6.5, but it does not cover the full functionality specified by the X3D specification yet.

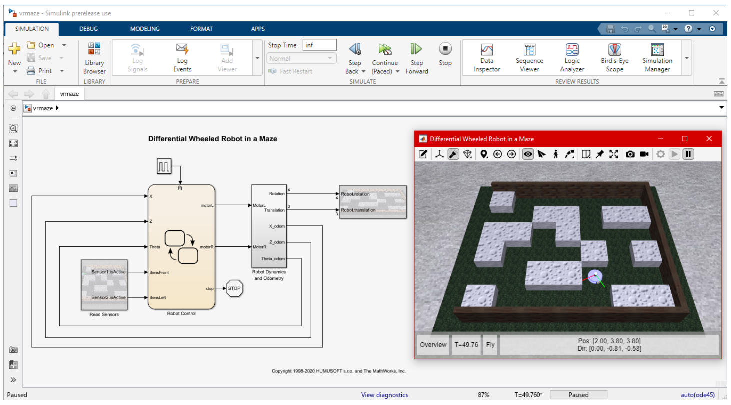

Differential Wheeled Robot in a Maze
Robot Control
Read Sensors
Robot Dynamics and Odometry
Robot.rotation
Robot.translation

Figure 6.4: Simulink 3D Animation *vrmaze* official example showing the functionality of the **LinePickSensor** node with the new renderer. In a scene with a maze, there is a robot with two sensors on its body. If a sensor detects a collision with a wall, the sensor visualisation beam changes its color to red and the robot acts according to this information.

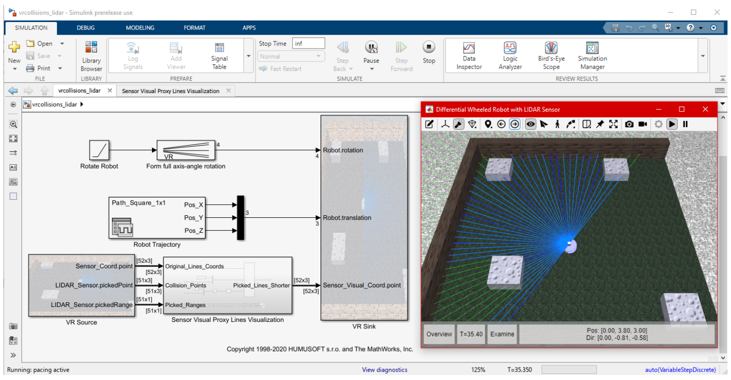

Differential Wheeled Robot with LIDAR Sensor
Rotate Robot
Form full axis-angle rotation
Robot Trajectory
VR Source
Sensor Visual Proxy Lines Visualization
VR Sink

Figure 6.5: Simulink 3D Animation *vrcollisions_lidar* official example showing the functionality of the **LinePickSensor** node with the new renderer. In a scene with a maze, there is a robot with many sensors on its body. If a sensor detects a collision with a wall, the point of collision is visualized by color of the sensor beam. Blue beam has not yet hit anything, green part of the beam is occluded by a wall.

6.5 Scene modification and interaction

Node addition, modification and deletion are fully implemented in the new Three.js-based renderer on all supported nodes. Upon testing with the official examples of Simulink 3D Animation, the performance of the renderer stayed consistent at around 60 FPS, apart from drops during heavy geometry modifications, which cause a slight drop in performance not only for the renderer but especially for the main process code during the needed scene traversal for new geometry export and transmission.

This drop is influenced not only by the number of modified geometries or the geometry size, but also by the dimensions of the scene itself.

Implemented user interaction includes scene navigation, anchor functionality and node selection for the virtual reality editor.

Another set of new features is accessible through virtual reality canvas properties: anti-aliasing, wireframe rendering, an object outline rendering for selection visualization and screen capture.

6.6 Comparison with Java based renderer

Although the new Three.js based renderer does not support all the features of the main Java dependent renderer, the performance of the two could be tested on the official Simulink 3D Animation shipping examples, that use features which are already implemented in the new renderer. The following tests were done on examples *vr_octavia*, *vr_octavia_2cars* and *vrcollisions_lidar*.

The tests were done through Simulink Profiler tool and it measured the time needed for simulation execution while there was no pacing that would try to synchronize the simulation time with the real time.

The examples *vr_octavia* and *vr_octavia_2cars* both require transformation modification of a high number of objects in the scene. Java based renderer performed much better in both of these tests, although it is important to note that the experimental renderer still performed faster than real time. The example *vrcollisions_lidar* requires ray casting in every simulation step and because Three.js implementation of ray casting is more optimized than the implementation in the main renderer, the experimental version performed better than the main version during the testing.

The stop time of the *vr_octavia* example was set to 111 seconds of simulation time and it used the sample time of 0.04 seconds. For the *vr_octavia_2cars* example, those values were 33 seconds stop time and 0.08 seconds sample time. The example *vrcollisions_lidar* had stop time set to 20 seconds and was sampled by 0.05 seconds.

Test results can be seen in the table 6.2.

	vr_octavia	*vr_octavia_2cars*	*vrcollisions_lidar*
Main version			
Round 1	48.8260	12.2660	19.8500
Round 2	48.5410	12.1900	20.0050
Round 3	48.2830	12.0960	20.2710
Round 4	48.2990	12.2310	20.2920
Round 5	48.3150	11.8050	20.3240
Round 6	48.3680	11.7890	20.2430
Round 7	48.4640	12.3130	20.3950
Round 8	48.3130	11.8670	20.2930
Round 9	48.2310	12.1310	20.3950
Round 10	48.3100	12.1310	20.6540
Average	48.3950	12.0819	20.2722
Experimental version			
Round 1	75.2820	15.0090	10.9480
Round 2	75.2060	14.2970	11.0810
Round 3	75.0320	13.9970	11.5490
Round 4	75.5690	13.8020	10.4930
Round 5	74.8880	14.0340	10.3190
Round 6	74.8020	13.9260	10.3180
Round 7	74.0300	13.8230	10.2900
Round 8	75.1330	14.0200	10.2710
Round 9	74.9310	14.1780	10.9080
Round 10	74.7080	13.8610	10.5030
Average	74.9581	14.0947	10.6680

Table 6.2: Results of tests done on the main and the experimental version of Simulink 3D Animation. Each example was run under a Simulink profiler tool. Examples used for testing are from official software examples *vrcollisions_lidar*, *vr_octavia* and *vr_octavia_2cars*. Time measurements are given in real-time seconds.

Chapter 7

Conclusion

Firstly, this work went over VRML, X3D and glTF file formats and scratched the surface of the capabilities of three 3D scene renderers.

X3DOM has shown great visuals, but many features Simulink 3D Animation needs are missing and its satisfying results with its current physics script have not been achieved.

Whereas X_ite does not yet have implemented everything needed for the same visual effect, their physics concepts were easy to get to work. Although it fully supports the **Immersive Profile** of X3D standard, that Simulink 3D Animation requires, it is under an unsuitable license, which makes it unfit for commercial use.

The most powerful library overall out of those three turned out to be Three.js, with big user contributions, many additional libraries to choose from and easy scripting. At the moment, its results in the field of visual quality and object physics go beyond both X3DOM's and X_ite's capabilities.

Upon exploring the current implementation of the software, modifications were proposed to the communication protocol, supported nodes and renderer itself.

Proposed modifications were then implemented, compared to the previous state of the software and results showcased.

The experimental version of Simulink 3D Animation is now able to render scenes of most of the official examples provided with the software. Its functionality was significantly enhanced as well as the spectrum of rendering features. Although throughout the implementation testing was done and the transition to a Three.js-based renderer has been fairly successful so far, some issues might still be raised during future development.

7.1 Future work

Transitioning to a Three.js-based renderer opened many possibilities for future work both in terms of rendering features but also with the collisions and physics.

During the implementation of this work, an idea was proposed regarding the physics engine to use, that it could be integrated either into the renderer or even directly into the internal scene. Ammo.js is a possibility for the first case, Bullet Physics, which Ammo.js is a port of, is a possibility for the second case.

Missing implementation of certain node types, mainly the rest of the X3D sensor nodes that the main version of the software currently supports will need to be implemented in the new renderer as well.

Support of glTF model loading is highly advised, not only is it a widely used file format, but it has also shown much better optimization for loading speed, which currently is a problem for detailed X3D models.

For the purposes of using the Three.js visual capabilities to the fullest and because the 4.0 version of X3D specification is still not published as a standard, Simulink 3D Animation will probably allow export of its own file format of scene description, which will be an enhanced variant of X3D file format.

Bibliography

[1] Web3D Consortium. *Information technology – Computer graphics and image processing – The Virtual Reality Modeling Language (VRML2) – Part 1: Functional specification and UTF-8 encoding.* 1997. https://www.web3d.org/documents/specifications/14772/V2.0/index.html

[2] Web3D Consortium. *Information technology — Computer graphics, image processing and environmental data representation — Extensible 3D (X3D) — Part 1: Architecture and base components.* 2013. https://www.web3d.org/documents/specifications/19775-1/V3.3/index.html

[3] Web3D Consortium. *Information technology — Computer graphics, image processing and environmental data representation— Extensible 3D (X3D) — Part 1: Architecture and base components.* 2022. https://www.web3d.org/documents/specifications/19775-1/V4.0/index.html

[4] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory To Implementation* https://www.pbr-book.org/

[5] S. Marschner and P. Shirley. *Fundamentals of Computer Graphics, 4th edition* CRC Press, 2016.

[6] The Khronos Group Inc. *glTF™ 2.0 Specification.* 2022. https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html

[7] X3DOM. *X3DOM main page.* https://www.x3dom.org/

[8] X3DOM. *X3DOM Documentation.* https://doc.x3dom.org/index.html

[9] X3DOM. *X3DOM Common Surface Shader tutorial and example.* https://doc.x3dom.org/tutorials/lighting/commonSurfaceShaderNode/index.html

[10] zixisun02. *glTF model "Shiba".* https://sketchfab.com/3d-models/shiba-faef9fe5ace445e7b2989d1c1ece361c

[11] Blender Foundation. *Blender.* https://www.blender.org/

[12] Don Brutzman, Andreas Stamoulias, Athanasios G. Malamos, Markos Zampoglou. *Enhancing X3DOM Declarative 3D with Rigid Body Physics Support.* 2014.

[13] CREATE3000. *X_ite main page* `https://create3000.github.io/x_ite/`

[14] CREATE3000. *X_ite tutorials* `https://create3000.github.io/x_ite/tutorials/overview`

[15] github.com/mrdoob. *Three.js main page* `https://threejs.org/`

[16] github.com/mrdoob. *Three.js documentation and tutorials* `https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene`

[17] Erwin Coumans. (2012) *Bullet 2.80 Physics SDK Manual.* `http://www.cs.kent.edu/~ruttan/GameEngines/lectures/Bullet_User_Manual`

[18] oxyn on Three.js forum *loading gltf meshes into ammo.js convex hull collision shape* `https://discourse.threejs.org/t/how-to-add-ammo-js-physic-to-gltf-file/27539`

[19] The MathWorks, Inc. *MATLAB documentation* `https://www.mathworks.com/help/matlab/index.html?s_tid=hc_panel`

[20] The MathWorks, Inc. *Simulink 3D Animation Official Documentation* `https://www.mathworks.com/help/sl3d/index.html?s_tid=CRUX_lftnav`

[21] The MathWorks, Inc. *C Matrix API, MATLAB documentation* `https://www.mathworks.com/help/matlab/cc-mx-matrix-library.html`

[22] A Tencent company, and Milo Yip. *RapidJSON, Main Page* `https://rapidjson.org/`

Appendix A
Abbreviations

3D	3-dimensional
IT	information technology
ISO	International Organization for Standardization
VRML	The Virtual Reality Modeling Language
UI	user interface
X3D	Extensible 3D
XML	Extensible Markup Language
PBR	Physically based rendering
IBL	Image-based lighting
glTF	Graphics Language Transmission Format
JSON	JavaScript Object Notation
BRDF	bidirectional reflectance distribution function
HTML	Hypertext Markup Language
DOM	document object model
URL	uniform resource locator
HDR	high dynamic range imaging
IOR	index of reflection
FPS	frames per second

Appendix B

Attached files index

```
/
├── imgs
├── src
│   ├── renderer
│   └── study
├── videos
├── "README"
└── "index.html"
```

- **img** - screenshots
- **src/renderer** - source code for attached renderer implementation
- **src/study** - source code for attached examples of JavaScript libraries study
- **videos** - videos demonstrating newly implemented functionality
- **README** - file containing information about the attached renderer source code
- **index.html** - page to be run demonstrating the study examples and basic renderer functionality