A New Visualization Framework for Simulink 3D Animation

Tereza Hlavová
Supervised by: Jan Houška

Department of Computer Graphics and Interaction
Czech Technical University in Prague
Prague / Czech Republic

Abstract

This paper focuses on visualization and interaction with a 3D scene in the Simulink 3D Animation tool for MATLAB. Our goal is to improve its visual quality, physics simulation capabilities, and performance. We propose a new rendering component using Three.js, a JavaScript 3D graphics library. We describe an implementation of the rendering component and its addition to the software. Compared to its predecessor, the new renderer supports some of the new visual features of the X3D format version 4.0, mainly physically based rendering (PBR), image-based lighting (IBL), and improvements in simple collision detection. We demonstrate the improvements and changes using official examples from Simulink 3D Animation.

Keywords: MATLAB, Three.js, Simulink 3D Animation, physical simulation, physically based rendering

1 Introduction

In the last decades, 3D graphics have come a very long way with new breathtaking improvements in visuals promised and realized every year. In contrast, a MATLAB software tool Simulink 3D Animation (SL3D) has been missing an up-to-date look, not receiving similar visual improvements in years. Features we wanted to focus on include physically based rendering (PBR), image-based lighting (IBL), and casting of shadows. PBR aims to represent an interaction between light and the surface of an object more accurately than empirical local illumination methods [12]. Under such model, objects with defined materials should look consistent under any lighting setup in the scene, which was not the case for the empirical model that SL3D used. IBL produces realistic reflections and ambient lighting from images and makes the objects appear as if they belong to a given environment. Casting of shadows can help better understand locations of lights and the scene and relative locations of 3D objects.

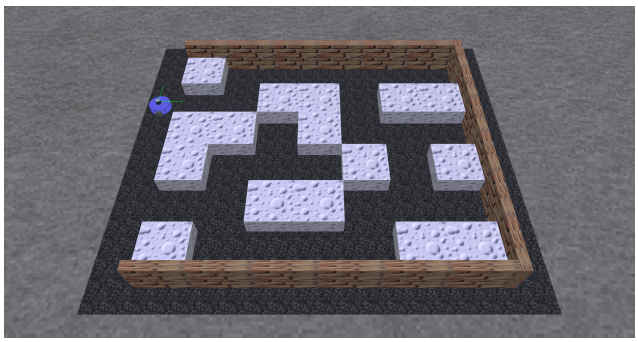
MATLAB development teams have been encouraged to transition their user interface components of Simulink and MATLAB tools using Java or other third-party technologies to web technologies. The main version of SL3D uses OpenGL 1.2.1 to render the scene and Java for its inclusion in the MATLAB graphical output displaying windows called figures. An experimental JavaScript-based branch of SL3D already existed before our work, so we analyzed its differences from the main version. The experimental version used a modified version of a JavaScript library called X3DOM for rendering 3D content [18]. X3DOM library does not support all the features that SL3D needs, mainly a *LinePickSensor* node needed for Simulink models imitating lidars, and therefore further modifications to the library were needed. Additions and modifications to the library are not trivial because it is declarative and the actual library functionality is mostly undocumented. The experimental version also performed poorly at stress tests frequently resulting at the scene not being loaded and object parameters not being updated in the scene.

We overviewed possible different directions of the development looking at three JavaScript 3D graphics libraries. We proposed a completely new implementation of its visualization component. We implemented a new renderer, interaction methods with the 3D scene and its inclusion to the rest of Simulink 3D Animation. Improvements in rendering quality are shown in Figure 1. The work was developed under HUMUSOFT s.r.o. for The MathWorks, Inc.

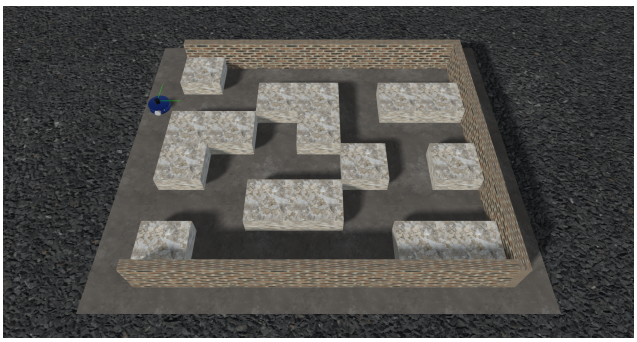
In section 2 we will describe SL3D and proposed modifications in more detail. The implementation is then overviewed in section 3 and the results are presented in section 4.

2 Background

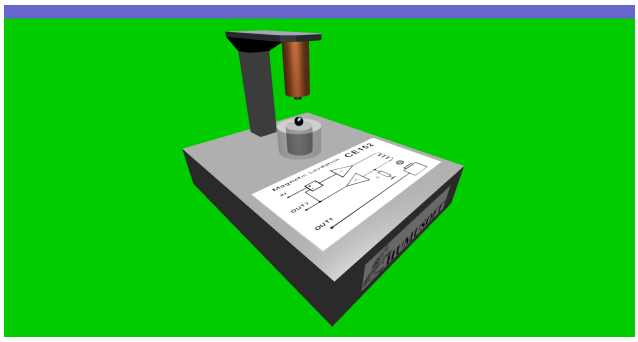
In this section, we will introduce the Simulink 3D Animation tool for MATLAB software, describe its use, implementation, and proposed goals and modifications for this work. MATLAB is a computing environment as well as a programming language developed by The MathWorks, Inc. It is widely used together with Simulink, a block diagram environment used to design systems with multidomain models, simulate before moving to hardware, and deploy without writing code [14].



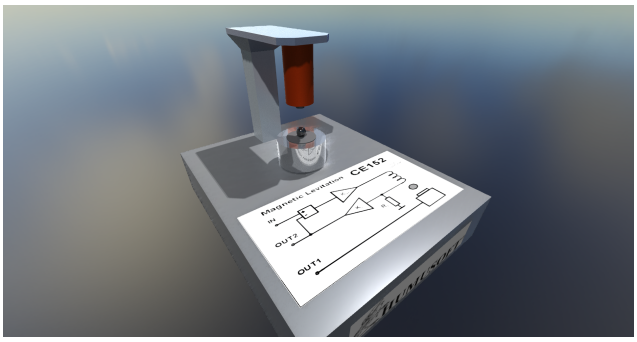
(a) *vrmaze* old example.



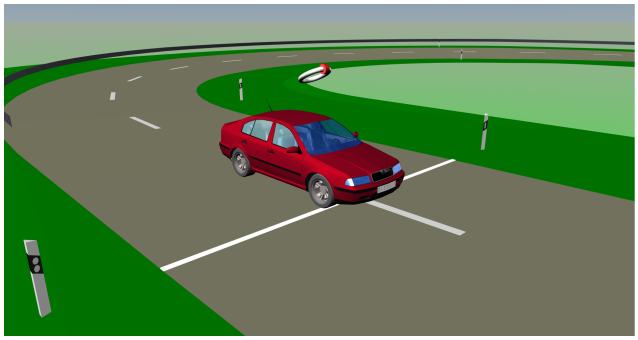
(b) *vrmaze* updated example.



(c) *vrmaglev* old example.



(d) *vrmaglev* updated example with SSR.



(e) *vr_octavia* old example.

(f) *vr_octavia* updated example.

Figure 1: Scenes from official Simulink 3D Animation examples, old lighting model versus using PBR, IBL and shadow casting.

2.1 Simulink 3D Animation

Simulink® 3D Animation™ is a tool under MATLAB software that links Simulink models and MATLAB algorithms to 3D graphics objects in virtual scenes [15]. With this tool, the user can load a 3D scene into a scene editor, modify its content, view the scene in a viewer, or connect attributes of the scene to those of a Simulink model to visualize the scene and its updates in a viewer.

SL3D in the MATLAB release version R2024a works with standardized scene formats VRML [3] and its successor Extensible 3D - X3D, version 3.3 [4]. It supports a list of features of the standard that are part of the Immersive Profile of X3D version 3.3. Both are declarative file formats describing 3D objects and scenes, as well as their behavior and user interaction. They are designed by the Web3D Consortium. The standards offer users a wide range of built-in scene node types for transformations, geometry, material definitions, and a prototyping concept used for creating new custom node types. To work with a 3D scene in Simulink, the tool offers library blocks. The blocks can write data into a 3D scene or read data from the 3D scene. This creates a connection between the fields of the scene nodes and the parameters in Simulink blocks.

The software tool can be divided into four main implementation parts:

- MATLAB interface,
- internal scene representation,
- canvases,
- editor/viewer.

The simplified software architecture outline can be seen in Figure 2. Most of the interface is implemented in MATLAB. The underlying functionality, scene storing and handling, is implemented in C++. There, the loaded 3D scenes are kept in an internal representation in classes loosely based on the OpenVRML Library [11]. C++ functions can be called from MATLAB functions with one main module called *vrclimex*. The other direction of communication is realized through callbacks. Users can either work directly with prepared functions or they are able to interact with the scene through virtual canvas classes, which maintain up-to-date modifiable scene properties.

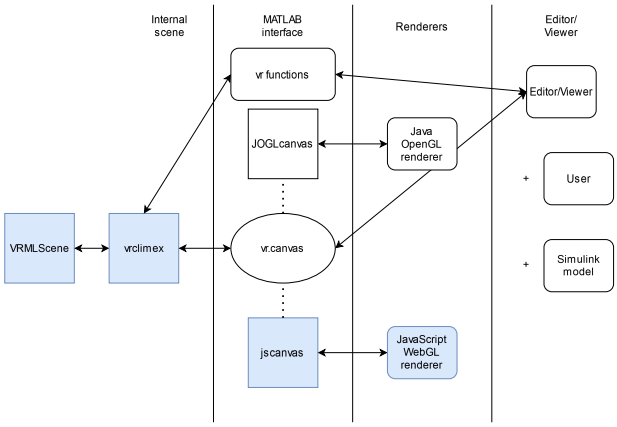


Figure 2: Simplified outline of Simulink 3D Animation architecture. The highlighted components were modified for the purposes of this work.

2.2 Proposed Modifications

The two SL3D versions take completely different approaches to the way the rendered images are produced and delivered to the user. The main branch encapsulates rendered frames directly to figures by performing traversal of the scene graph loaded into the internal scene representation. The experimental branch maintains its own separate renderer running in MATLAB's *HTML UI component*. This component internally uses a Chromium browser and displays HTML5 and JavaScript content. The *HTML UI component* runs independently from the MATLAB processing and offers a JSON-based communication channel without any synchronization guarantees. We performed stress tests on the experimental version before our implementation. In these tests, we periodically created a new canvas and checked for correctly loaded scene properties in the MATLAB canvas. Not a single instance out of 50 ensured the correct load of the chosen scene viewpoint. Testing an animation of a sphere object on a circular trajectory without any additional wait resulted in 1633 ignored animation steps out of 2000. Based on that we concluded that a communication protocol is needed to ensure the delivery of scene updates and user requests between the MATLAB interface and the renderer in the *HTML UI component*.

For the rendering component itself, we considered open-source 3D graphics libraries: the previously used X3DOM, X_ite [7], and Three.js [10]. The license of X_ite demands the source code of the software using it to be publicly available and thus is not fit for commercial use. X3DOM does not offer flexibility for the implementation of missing features or modifications in general. We decided to completely re-implement the experimental branch, adding a new renderer using the Three.js library. It offers control over scene building, animating, and rendering and thus also the flexibility that X3DOM lacks. Possible support of real-time physics simulation in SL3D could be also added this way in the future because Three.js is capable of including ammo.js [9], which is a direct port of Bullet Physics Engine [6] into JavaScript. X3DOM offers a physics simulation component [1] too, but there is no user documentation, no official examples and we were not able to produce working examples ourselves during testing.

Rendering VRML and X3D version 3.3 worlds only according to their standards would not use the wide variety of features Three.js is capable of. In order to massively enhance the visual capabilities of SL3D we suggested implementing new features also in the internal scene representation. A new X3D standard version 4.0 [5] was approved in December 2023. Its additions and changes might be crucial for this and future work. Possibly the most important addition is the inclusion of PBR through *PhysicalMaterial* node and shadow casting through *castShadow* field in a *Shape* node. We finished this work before the official finalization of the standard version when also an *EnvironmentLight* node was still a part of the standard as well before being removed for the finalized release, which included IBL.

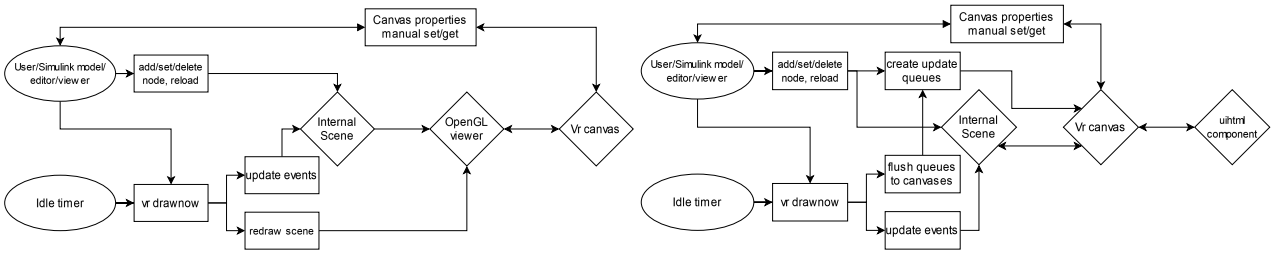
3 Implementation

This section will describe the principles of the implementation of the most important modification decisions.

3.1 Architecture Changes

After solving X3D format version control in the internal scene representation, new nodes and also new fields enhancing the existing nodes had to be marked accordingly. After these modifications, the software was capable of loading nodes and fields required for the Immersive profile of X3D 4.0 specification into the internal scene.

A message ID confirmation system was needed to ensure that no exchanged information was outdated. We implemented a new communication protocol between the renderer's code running in the *HTML UI component* and the rest of the architecture - mainly regarding updates from the internal scene but also requests from and to the MATLAB interface.



(a) Standard Java-based Simulink 3D Animation.

(b) Experimental Simulink 3D Animation, newly implemented.

Figure 3: Modification of scene and canvas properties propagation.

In the main branch, the propagation of scene modifications into the rendered frame was ensured by calling a *drawnow* function. It executes and flushes all scene modification calls. It is usually called by Simulink after every simulation step or by the editor upon user interaction. For updates in the scene event system, an internal idle timer also executes the calls on a periodic basis. This propagation of scene modification to the viewport in the main branch is visualized in Figure 3a.

For the experimental branch, we implemented a virtual canvas registration process for scene updates. The updates get stored in update queues. Upon a *drawnow* call, an update message is produced from all and sent back to all canvases registered for the scene. The scene modification propagation to the viewport in the new experimental version can be seen in Figure 3b.

The scene has to be exported for the renderer of the experimental branch because it is run separately. For messages that describe a scene or part of the scene, we used encoding into JSON format using a library called RapidJSON [2]. Scene nodes are mapped to objects, and field to object properties. The initial scene export into JSON format is built using a modified internal scene traversal which effectively prints out all information deemed important for the functionality of the external renderer. Scene modification updates created upon a *drawnow* call also use the same export functionality but are limited to the nodes they relate to.

3.2 JavaScript Renderer

As stated before, we implemented the Javascript renderer using the Three.js library. The main script starts after MATLAB calls its initialization function. It sets up the communication protocol control and constructs a rendering pipeline.

3.2.1 Scene Import and Updating

Three.js is not originally meant to work with VRML or X3D files. Its own scene representation and overall library functionality differ from the said file format standards. It was important for us to use as much already existing functionality of this powerful library to fit the standards, but even with those efforts many node types did not have matching Three.js equivalents, or at least not entirely.

The scene maintained in the renderer has to accept updates from the internally maintained one while also following the VRML mechanisms of reusing nodes. Thus we decided to use two main structures in the renderer. There is a scene graph, which is rendered by Three.js, and all its nodes are inherited from Three.js classes. Then there is a map of nodes that holds references to all instances under every node ID. Both structures are held in a scene script that is also responsible for managing scene navigation, user interaction, and rendering settings propagation to the scene nodes. It also provides a map of functions for node building and updating out of the JSON representation.

Similarly to the internal scene class inheritance hierarchy, the JavaScript renderer code defines a class for every supported scene node and the building function generates their instances. Each node always implements a constructor, a *clone* method possibly with a *copy* method, an *init* method for creation, a *set* method for non-default values during both scene import and scene updates, and a *delete* method for proper disposal of resources both locally and on the GPU. The classes either derive from native Three.js classes, enhancing them to fit X3D concepts, or were implemented anew.

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 X3D's *Shape* nodes. While the X3D standard specifies a *Shape* node capable of holding any kind of geometry node and appearance descriptions, Three.js provides different scene objects for different geometry types they are able to present. This forced us to divide some of the X3D node definitions from objects actually used in the scene and keep them both, which allowed for better control over shared resources and scene updates.

The most important classes that hold X3D definitions and manage separate Three.js objects are shape, all geometry classes, and appearance classes containing material classes and texture classes. All of them influence values in (usually multiple) Three.js objects that actually play a part in the scene rendering and manage re-referencing and reusing the resources, or their disposal when they are not

used anywhere anymore.

A source code example for a material update can be seen below. It implements base/diffuse texture addition to all required real materials in the Three.js shape objects:

```
// 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 using them
  for (const app of x3dmat._x3dappearances)
  { // apply appearance's texture transform
    app.setTextureTransform(newmap);
    const mats = app.getRealMaterials();
    // and apply it to all real materials
    // linked to that appearance's shape
    for (const mat of mats)
    {
      if (mat.map) // remove old texture
        mat.map.dispose();
      mat.map = newmap.clone();
      mat.needsUpdate = true;
    }
  }
  newmap.dispose();
}
```

The implementation of the text rendering in particular is non-trivial. This is due to the X3D standard being more flexible than the text rendering methods offered by Three.js (mainly the alignment, justification, direction, UTF-8 fonts, and material application requirements). We implemented it by rendering the text into texture from an HTML canvas element, which we deemed flexible enough. The texture is used as a mask on a plane shape which can have a material applied to it. An example can be seen in Figure 4. The downside of this approach is that the resolution for the rendered texture is pre-set. Making this dependent on the position of the camera could be a topic for future work.

The new renderer's supported nodes' mapping to Three.js classes can be seen in Table 1. Geometry gets already triangulated in the internal scene, originally for OpenGL, X3D geometry nodes missing in the table are exported for the renderer as pre-triangulated *IndexedFaceSet*. Supported sensors are also not included in the table, they do not derive from any Three.js class and are discussed in the following paragraphs.

Three.js has an inbuilt loader for models of glTF format [13]. We have also allowed the inclusion of such models in inline nodes using the loader. However, their properties cannot be modified, because the internal scene does not work with this format yet. The models are just inserted into the scene and are influenced by the transformation hierarchy.

Table 1: Overview of implementing X3D nodes using Three.js.

X3D Node	Three.js class	relationship
Networking Component		
Anchor	Group	inheritance
Inline	Group	inheritance
Grouping Component		
Group	Group	inheritance
Switch	Group	inheritance
Transform	Group	inheritance
Rendering Component		
IndexedFaceSet	BufferGeometry	inheritance
IndexedLineSet	BufferGeometry	inheritance
Shape Component		
Material	MeshPhongMaterial, LineBasicMaterial, MeshUnlitMaterial	encapsulation
PhysicalMaterial	MeshStandardMaterial, LineBasicMaterial, MeshUnlitMaterial	encapsulation
Shape	Mesh, LineSegments	encapsulation
Geometry3D Component		
Box	BoxGeometry	inheritance
Cone	BufferGeometry	inheritance
Cylinder	BufferGeometry	inheritance
Sphere	SphereGeometry	inheritance
Text Component		
Text	PlaneGeometry	inheritance
FontStyle	[none]	–
Lighting Component		
EnvironmentLight	Scene	parameter
DirectionalLight	DirectionalLight	inheritance
SpotLight	SpotLight	inheritance
PointLight	PointLight	inheritance
Texturing Component		
ImageTexture	Texture, CanvasTexture	encapsulation
TextureTransform	[none]	–
Navigation Component		
Billboard	Group	inheritance
NavigationInfo	Object3D	inheritance
Viewpoint	Object3D	inheritance
Navigation Component		
Background	Mesh/Scene	encapsulation /parameter

3.2.2 Sensor Nodes Functionality

Sensor nodes we have implemented in the new renderer so far include *ProximitySensor*, *TouchSensor*, *PlaneSensor*, *LinePickSensor*, and *PrimitivePickSensor*. When an enabled sensor is called to update on every simulation step, it evaluates its state and when needed, adds its output information to a message queue that gets sent over to MATLAB after all sensors are evaluated. The messages are then processed in the internal scene representation to be available for reading by the user or Simulink model. Only one can-

vas is chosen as the main and is responsible for producing and sending back possible sensor updates.

We have used simple collision detection and intersection computation functions provided by Three.js for the sensor activity evaluation. That includes ray-casting into the scene, or triangle/sphere and triangle/box intersections. A special case was *LinePickSensor*, which uses line geometry to detect hits with a chosen subtree of scene graph objects. The line geometry does not have to be made out of individual straight lines for every sensor but instead can be a general line geometry with many line segments. Thus we process every line segment individually during the update. The pseudocode for processing an individual segment can be seen below. *LinePickSensor* node in use is shown in Figure 5.

ALGORITHM 1
Sensor line segment processing algorithm

$startPoint \leftarrow parent$.localToWorld($startPoint$)
$endPoint \leftarrow parent$.localToWorld($endPoint$)
$length \leftarrow startPoint$.distanceTo($endPoint$)
$direction \leftarrow endPoint$.clone().sub($startPoint$)
$direction$.normalize()
$raycaster$.set($startPoint$, $direction$)
$raycaster.far \leftarrow length$
$intersects \leftarrow raycaster$.intersectObjects(...$target$)

Figure 4: Text rendering for the official example *vr_panel*.

3.2.3 User Interaction and Navigation

A viewpoint binding mechanism was implemented directly according to the X3D specification. We wanted to ensure quick and usable methods of navigation in the scene, so we decided to implement the following principles:

- 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.
- The camera is grounded under the WALK navigation type of X3D specification.

Apart from navigation and some of the sensors, user is also able to interact with the scene in edit mode. In this mode, the sensor functionality is postponed, and picking interaction is instead evaluated as node selection through ray-casting. The clicked nodes in the scene get highlighted and their fields are revealed in a world editor to modify as shown in Figure 6.

3.2.4 Rendering Pipeline

For post-processing management, we have used post-processing add-on to Three.js developed by Raoul van Rüschen [17]. Apart from the normal render pass, outline pass for highlights is added in edit mode. Subpixel morphological anti-aliasing pass can be added through canvas settings. A custom screen capture pass renders to a buffer on screenshot request. As an experimental feature, we have also added screen space reflections implementation from Realism Effects developed by 0beqz [8] in two additional passes - *velocityDepthNormal* pass and *SSREffect* pass.

4 Results

Most of the examples used for testing, comparisons, and showcase are the official examples of Simulink 3D Animation [16].

4.1 Communication Protocol

Before our work, the experimental X3DOM-based viewer did not use any message confirmation protocol and did not guarantee to offer up-to-date information on its virtual reality canvas properties. 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 and also an inability to reasonably measure performance.

Now, the implemented communication protocol does guarantee waiting on load events, and confirmation of messages when a renderer-dependent property value is requested. Due to the object nature of the new protocol, screen capture image data transfer from the renderer to the canvas was made possible and implemented as well. The same communication stress tests, which failed before the implementation, all passed with this new implementation.

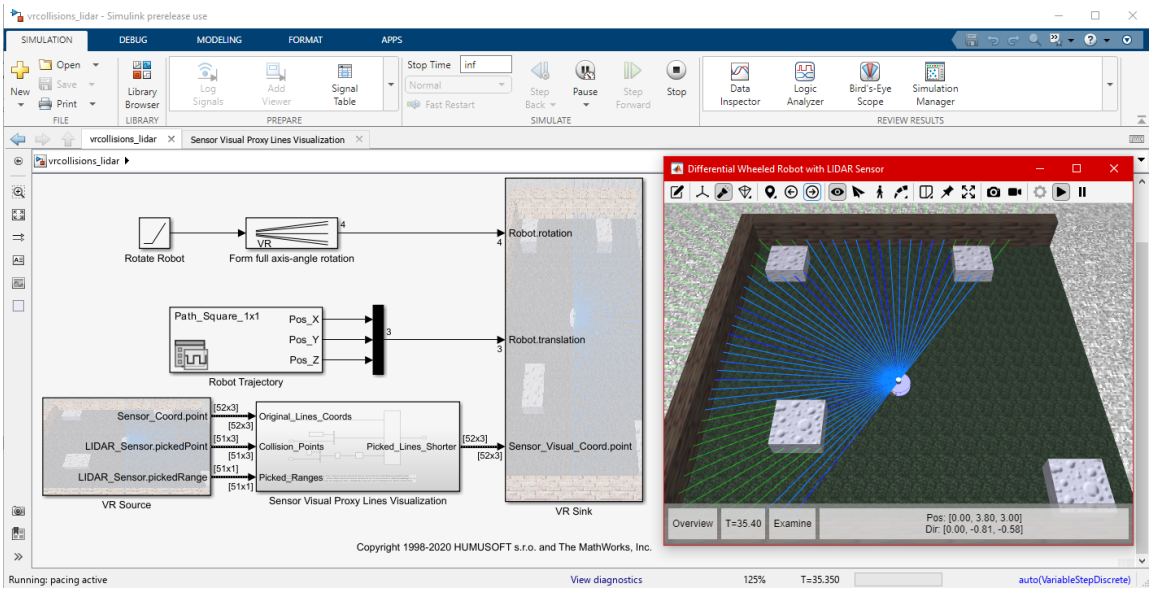


Figure 5: Simulink 3D Animation *vrcollisions_lidar* official example showing the functionality of the *LinePickSensor* node with the new renderer. There is a robot with many sensors on its body. If a sensor detects a collision with a wall, the point of collision is visualized by color of the sensor beam. The blue beam has not yet hit anything, the green part of the beam is occluded by a wall.

Figure 6: Example of highlights rendering for editing mode made with Outline rendering pass.

4.2 Supported Nodes

Current scene export and import for the experimental renderer supports most of the nodes the internal scene by itself does with the exceptions of points-related nodes, a *MovieTexture* node, a *PixelTexture* node, sound nodes, and the rest of the sensor nodes, which have not been implemented yet. Most of the scenes used by Simulink 3D Animation official examples are able to fully load with minor visual differences. Loading times of the scenes vary, the official examples taking a maximum of seconds, but scenes heavy on detail with millions of vertices do not get loaded in a reasonable time, similar to how the previous X3DOM-based viewer performed, or even the standard Java-based viewer in some cases does.

Upon modification and enhancement of the internal scene's supported node types and implementing support for given scene node representation in the JavaScript-based renderer, new visual features are now possible to use. This has allowed us to update old official Simulink 3D Animation examples, comparisons are shown in Figure 1. Performance measurements can be seen in Table 2. The example *vr_octavia* is simpler than the other two in number of animated objects and sensor activity. From the visible difference between FPS of Java-based viewer and JavaScript-based viewer in this example we can conclude that the bottleneck of the process is the implemented communication protocol and communication channel of *HTML UI component*. The Java-based viewer not limited by the communication management performs better.

Table 2: Results of tests done on the standard 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*. Performance measurements are given in frames per second.

Example	**Java-based viewer**	**Three.js viewer**
vr_octavia	75	44
vr_octavia_2cars	44	44
vrcollisions_lidar	40	44

LinePickSensor node functionality needed for Simulink

3D Animation official examples of *vrmaze* and *vrcollisions_lidar* is fully implemented in the experimental version. It does not cover the full functionality specified by the X3D specification yet, but it does improve on the main version of SL3D. The intersection computation using Three.js ray-casting is more precise than that of the main version of SL3D where only intersections with bounding boxes and bounding spheres are implemented.

5 Conclusions and future work

Upon exploring the current implementation of Simulink 3D Animation, we proposed modifications to the communication protocol, supported nodes, and renderer itself. We implemented the changes, compared the software tool to its previous state, and showcased the results.

The experimental version of SL3D is now able to render the scenes of most of the official examples provided by 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 new renderer based on the library Three.js has been fairly successful so far, some issues might still be addressed during future development.

Missing implementation of certain node types will need to be implemented in the new renderer as well. For the purposes of using the Three.js visual capabilities to the fullest, Simulink 3D Animation will probably allow exporting of its own file format of scene description, which will be an enhanced variant of the X3D file format.

In the future, it could be interesting to integrate a full physics engine either into the renderer or even directly into the internal scene representation in MATLAB.

Acknowledgments

This work was supported by the Grant Agency of the Czech Technical University in Prague, No SGS22/173/OHK3/3T/13.

References

[1] Don Brutzman, Andreas Stamoulias, Athanasios G. Malamos, and Markos Zampoglou. Enhancing x3dom declarative 3d with rigid body physics support. 2014.

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

[3] Web3D Consortium. *Information technology – Computer graphics and image processing – The Virtual Reality Modeling Language (VRML2) – Part 1: Functional specification and UTF-8 encoding.*, 1997.

[4] Web3D Consortium. *Information technology — Computer graphics, image processing and environmental data representation— Extensible 3D (X3D) — Part 1: Architecture and base components.*, 2013.

[5] Web3D Consortium. *Information technology — Computer graphics, image processing and environmental data representation— Extensible 3D (X3D) — Part 1: Architecture and base components.*, 2022.

[6] Erwin Coumans. *Bullet 2.80 Physics SDK Manual.* http://www.cs.kent.edu/∼ruttan/GameEngines/lectures/Bullet_User_Manual.

[7] CREATE3000. *X_ite X3D Browser*. https://create3000.github.io/x_ite/.

[8] github.com/0beqz. *Realism Effects for Three.js.* https://github.com/0beqz/realism-effects.

[9] github.com/kripken. *Ammo.js - Direct port of the Bullet physics engine to JavaScript using Emscripten.* https://github.com/kripken/ammo.js.

[10] github.com/mrdoob. *Three.js Official Documentation*. https://threejs.org/docs/index.html.

[11] Chris Morley and Braden McDaniel. *OpenVRML.* https://sourceforge.net/projects/openvrml/.

[12] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory To Implementation*. The MIT Press, 4 edition, 2023.

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

[14] The MathWorks, Inc. *MATLAB Documentation*. https://www.mathworks.com/help/matlab/index.html.

[15] The MathWorks, Inc. *Simulink 3D Animation Documentation*. https://www.mathworks.com/help/sl3d/classic-virtual-reality-world.html.

[16] The MathWorks, Inc. *Simulink 3D Animation Official Examples.* https://www.mathworks.com/help/releases/R2023a/sl3d/examples.html.

[17] Raoul van Rüschen. *Post Processing for Three.js.* https://pmndrs.github.io/postprocessing/public/docs/.

[18] X3DOM. *Official X3DOM Documentation.* https://doc.x3dom.org/gettingStarted/index.html.