x3ogre: connecting X3D to a state of the art rendering engine

Pavel Rojtberg
pavel.rojtberg@igd.fraunhofer.de
Fraunhofer IGD

Benjamin Audenrith
benjamin.audenrith@igd.fraunhofer.de
Fraunhofer IGD

ABSTRACT

We connect X3D to the state of the art OGRE renderer using our prototypical x3ogre implementation. At this we perform a comparison of both on a conceptual level, highlighting similarities and differences. Our implementation allows swapping X3D concepts for OGRE concepts and vice versa. We take advantage of this to analyse current shortcomings in X3D and propose X3D extensions to overcome those.

CCS CONCEPTS

• **Computing methodologies → Graphics file formats**;

KEYWORDS

X3D, webgl, materials, geometry

(a) Per-pixel lighting   (b) Night vision compositor effect

Figure 1: The flipper.x3d example showcasing some of the OGRE features

1 INTRODUCTION

X3D (Daly and Brutzman 2007) is an open standard for 3D graphics with precisely defined semantics. Scenes stored in the X3D format can be parsed using standard XML parsers and the files are usually self-contained which makes X3D a good choice for interchange. However the standardization process causes that new rendering techniques and concepts appear in X3D with a considerable delay. The available X3D based rendering engines like X3DOM (Behr et al. 2009) or InstantReality (Fraunhofer IGD 2016) therefore offer custom extensions to overcome this. Yet those are only scarcely used as they impede interchange.

This paper therefore takes a different approach and instead connects X3D to an existing state of the art rendering engine. This allows using the file formats of the underlying renderer where X3D falls short. As these formats neither are standardized nor have to provide legacy compatibility, they can evolve faster and at the same time are better optimized for rendering. In comparison with creating an X3D extension this approach is more flexible as it allows replacing even core X3D concepts. This flexibility in turn gives better insights on how to improve X3D itself.

Here, we focus on the presentation aspect of X3D as it is arguably the most important part of a 3D file format. When using the X3D DOM profile (Behr et al. 2009) most application logic is outside of the X3D format and recent extensions (e.g. (Schwenk et al. 2012)
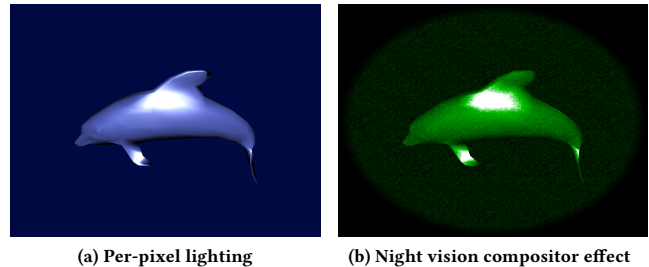
and (Sturm et al. 2016)) specifically target material representation. Therefore, we neglect the Scripting and Sensor parts of X3D and concentrate the Rendering and Geometry components.

For the underlying renderer we chose the Object-Oriented Graphics Rendering Engine (OGRE) (Streeting 2012) . While other rendering engines like Unreal 4 and Unity recently gained more popularity, OGRE is available royalty-free under an permissive open-source license making it a better fit for research as well as allowing deeper inspection.

OGRE is not bound to a specific rendering API like OpenGL or DirectX, but rather provides high level concepts that map to any of those. It is being developed since 1999 and was used in AAA games like Torchlight 1/2 as well as in industrial training applications. Therefore the rendering concepts are mature and proven — while having been developed independently to X3D. This makes a comparison especially interesting.

The comparison is performed using our prototypical implementation called "x3ogre", which allows loading X3D scenes in OGRE as well as using OGRE concepts inside X3D. Besides the comparison, one use-case of the prototype is to visually enhance existing X3D scenes without requiring invasive changes.

By utilizing Emscripten (Zakai 2011) OGRE based applications can be deployed on the Web[1]. Internally a GLES3 (Khronos 2016) based renderer is used, which supports WebGL (Khronos 2017) while also providing forward compatibility with WebGL2. Therefore the results are not limited to the C/ C++ ecosystem, but can also be transferred to the web context.

This paper is structured as follows: in section 2 the mapping of the X3D concepts to the corresponding OGRE concepts is discussed, while section 3 takes the reverse way, describing the integration of OGRE concepts in X3D. Based on the preceding discussions we then propose several X3D extensions in section 4. Finally we conclude with section 5 giving a summary of our results and discussing the limitations and future directions.

---

[1]https://ogrecave.github.io/ogre/emscripten/

2 TRANSFERRING X3D CONCEPTS TO OGRE

As the first step we identify the concepts corresponding to X3D in OGRE. Here we focus on a subset of the X3D interchange profile (ISO/IEC 19775-1 2012) which roughly corresponds the X3D DOM profile introduced by (Behr et al. 2009).

We will briefly discuss the high level objects and then focus on the compound geometry and material objects in more detail.

The correspondences for the core X3D objects are:

- the X3D Scene node corresponds to the *SceneManager*. In X3D only one Scene node per file is allowed, while OGRE supports several active SceneManager instances. With only one of them being rendered by a specific Camera.
- the X3D Transform node corresponds to two nested *SceneNode*s. SceneNodes only store a translation vector, an orientation quaternion and a scaling vector. Therefore two SceneNodes are needed to support the *center* property of a Transform. The X3D *scaleOrientation* property can only be implemented for multiples of $90°$ as there is no shearing component in OGRE.

  Note that OGRE does not have a native serialization format for SceneNodes and thus benefits by using X3D here.
- the X3D Geometry node corresponds to the *Mesh* object. As in X3D this is a compound object. It can be serialized either as XML or in a binary file format.
- the X3D Appearance node corresponds to the *Material* object. Again this is a compound object. It is serialized in a custom file format resembling the VRML97 encoding.

Additionally we identified the following correspondences needed to support animations. These are given for completeness as the concepts mostly map one-to-one:

- the X3D TimeSensor maps to *AccumulateControllerFunction*.
- the X3D ScalarInterpolator maps to *LinearControllerFunction*.
- the X3D CoordinateInterpolator maps to *VertexAnimationTrack*.
- the X3D PositionInterpolator and OrientationInterpolators correspond to *NodeAnimationTrack*.

Note that for supporting animations in X3D one does not need to implement the full X3D event model. Instead it is sufficient to rewrite specific ROUTE statements in a compositional way as shown in listing 1.

```xml
<TimeSensor DEF="time" />
<PositionInterpolator DEF="move" />
<ROUTE fromNode="time" fromField="fraction_changed"
       toNode="move" toField="set_fraction" />
<!-- is transformed into -->
<PositionInterpolator DEF="move">
  <TimeSensor USE="time" />
</PositionInterpolator>
```

Listing 1: Compositional ROUTE implementation

This approach requires allowing specific source nodes as children of the target nodes and is therefore not compatible with general ROUTE statements. However it enabled the correct loading of the existing X3D scenes we tried.

2.1 Geometry

OGRE uses the binary .mesh file format which can be transparently converted from and to XML for inspection and debugging.

Listing 2 shows a typical X3D geometry definition which translates to the OGRE XML format in listing 3.

```xml
<Shape>
  <Appearance USE='Example' />
  <IndexedTriangleSet coordIndex="0 1 2 [...]">
    <Coordinate point="0 0 0 [...]" />
    <Normal vector="0 0 0 [...]" />
  </IndexedTriangleSet>
</Shape>
```

Listing 2: Sample X3D Shape definition

The Shape node corresponds to a submesh node and the IndexedTriangleSet node corresponds to geometry node. Note that OGRE allows interleaved storage of the position and normal vertex attributes while X3D does not. This is useful, as OGRE also stores additional vertex attributes like bone assignments for skeletal animation (HAnimSegment node in X3D) in the .mesh file.

```xml
<submesh material="Example"
         usesharedvertices="false"
         operationtype="triangle_list">
  <faces count="815">
    <face v1="1" v2="2" v3="3" />
    [...]
  </faces>
  <geometry vertexcount="531">
    <vertexbuffer positions="true" normals="true">
      <vertex>
        <position x="0" y="0" z="0"/>
        <normal x="0" y="0" z="0"/>
      </vertex>
      [...]
    </vertexbuffer>
  </geometry>
</submesh>
```

Listing 3: Sample OGRE XML Mesh definition

Listing 3 shows a single submesh definition, but generally a .mesh file stores multiple submeshes. This allows the definition multi-material meshes, where each submesh is rendered with one material. In X3D one has to use multiple Shapes and then use a Group node for linking them.

Furthermore the submeshes can reference the same vertices to avoid data duplication. This representation directly maps to the low-level graphics APIs. (In OpenGL: *glDrawRangeElements* and *glBufferData*).

For X3D this concept can be reproduced by DEF/ USE of the Coordinate node — yet due to the complexity of the system it is not obvious whether X3D viewers will share the memory between the corresponding Shapes.

In OGRE, used material can only be referenced in the .mesh file while in X3D it is typically defined inline with the geometry data. This separation is similar to the concept in glTF (Robinet and Cozzi 2013) or the X3D BinaryGeometry node (Behr et al. 2012). Mesh files do not specify any compression. However OGRE assets are usually distributed in zipped packs containing geometry, materials and textures which offer compression on a higher level.

2.2 Materials

OGRE uses the custom .material script format for material definition which resembles the classic VRML encoding. For comparability we will use the VRML encoding for the following X3D material examples.

```
DEF Example Appearance {
  material Material {
    ambientIntensity 0.508497
    diffuseColor 0.337255 0.4 0.788235
    specularColor 1 1 1
} }
```

Listing 4: Example X3D Appearance definition

The X3D material in listing 4 translates to the OGRE material given in listing 5.

```
material Example {
  technique {
    pass {
      ambient 0.508497 0.508497 0.508497
      diffuse 0.337255 0.4 0.788235
      specular 1.0 1.0 1.0 25
} } }
```

Listing 5: According OGRE Material definition

OGRE materials support multiple techniques which are again composed of several rendering passes. The technique range allows picking the appropriate one at runtime based on hardware support, LOD level etc., while defining multiple passes can be useful for advanced rendering techniques like rendering hair.

Both OGRE and X3D material definitions reflect the simple Blinn-Phong shading model (Blinn 1977). However state of the art rendering usually involves more sophisticated lighting models like the Cook-Torrance microfacet reflection model (Cook and Torrance 1981) — optionally combined with normal mapping and deferred shading.

This requires a more flexible material definition. (Schwenk et al. 2010) therefore introduced the X3D CommonSurfaceShader node that used the uber-shader (Hargreaves 2005) approach. While offering more flexibility then the traditional materials, the monolithic nature requires the change of existing materials whenever a new rendering technique must be incorporated; the CommonSurfaceShader had to be updated to incorporate Physically Based Shading (PBS) (Schwenk et al. 2012).

In contrast OGRE provides the high level material system (HLMS) for defining custom materials. This system builds around the idea of passing opaque properties to a named template shader.

```
<Appearance DEF="Example">
  <PhysicalMaterial albedoFactor="0.22 0.3 0.5"
    roughnessFactor="0.4",
    metallicFactor="0.76" />
</Appearance>
```

Listing 6: Physically based material in X3D

For instance the PhysicalMaterial node of (Sturm et al. 2016) (listing 6) translates to the following HLMS material in OGRE.

```
hlms Example PBS {
  diffuse   1 1 1
  specular  1 1 1
  roughness 0.4
  fresnel   1.3
```

```
}
```

Listing 7: Physically based material in OGRE

However while (Sturm et al. 2016) rely on a predefined Shader, OGRE just forwards the given parameters to a shader named "PBS" (HLSL on DirectX , GLSL on OpenGL). This allows users to define custom materials with arbitrary parameters. Figure 2a shows a grid of spheres rendered with custom PBS shading while the ground plane is being rendered using the classical Blinn-Phong shading.

3 CONNECTING OGRE TO X3D

In Section 2 we identified the X3D concepts inside OGRE which allowed loading X3D files. This section on the other hand will focus on bringing the OGRE concepts to X3D. First we describe the connection for concepts which exist in both X3D and in OGRE, where the OGRE counterparts usually offer more flexibility. Then we will describe the mapping of the OGRE compositor system for which X3D has no counterpart.

3.1 Connection of concepts

The general approach taken here is to redirect existing X3D concepts to their OGRE counterparts as identified in section 2. In contrast to creating new X3D nodes, this simplifies the implementation and allows to first evaluate the benefits before introducing new concepts in X3D. For instance to use the interleaved vertex attribute storage in X3D we just redirect the Geometry node to an OGRE .mesh file and override the material by the X3D appearance. Listing 8 shows the different redirection options we support in x3ogre.

```
<!-- OGRE defined mesh, may reference X3D materials -->
<Shape USE="Sinbad.mesh" />
<!-- OGRE defined material in a X3D Shape -->
<Shape>
  <Appearance USE='Ogre/ExampleMaterial' />
  <IndexedTriangleSet>[...]</IndexedTriangleSet>
</Shape>
<!-- OGRE defined geometry in a X3D Shape -->
<Shape>
  <Appearance>[...]</Appearance>
  <Geometry USE="Sinbad.mesh" />
</Shape>
```

Listing 8: Using OGRE formats inside X3D

Both, the X3D DEF/ USE system and the OGRE resource system rely on strings for identifying resources. The implementation therefore is straightforward and we can maintain compatibility with existing X3D files by only using OGRE resources if the USE lookup inside of the current X3D file fails.

Compared to the ExternalGeometry node proposed by (Limper et al. 2014) this approach is easier to implement and does not requires invasive changes to existing X3D files. However it implicitly relies on an externally defined resource pool and overriding individual mesh properties (e.g. color) is not possible.

3.2 Explicit compositing

A compositing system allows specifying full screen effects to be applied to the image after a scene has been rendered — similar to layer effects in an image editing software. A simple example would be desaturation to create a black & white image without having to
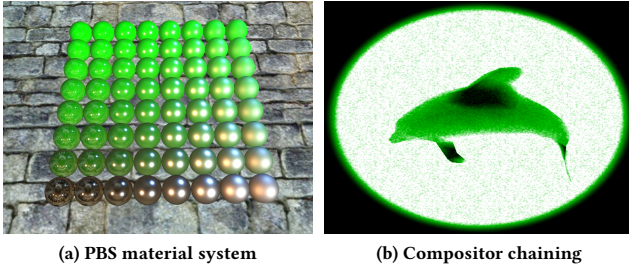
(a) PBS material system   (b) Compositor chaining

Figure 2: Advanced rendering concepts

change the scene materials. Figure 1b shows a more complex effect that involves adding noise and vignetting at the image borders.

X3D does not have an explicit compositing concept. Post processing is usually implemented by redirecting the rendering to a RenderedTexture which is then rendered using a custom Appearance on a full-screen quad. Combining multiple layers is only possible using the MultiTexture node. However it only offers a fixed set of blending modes that correspond to an OpenGL 1.2 extension (Khronos 2006). Chaining post processing effects is not feasible as there is no mechanism in X3D to specify in which order RenderedTextures should be processed.

OGRE on the other hand uses the explicit .compositor file format (see listing 9) that resemble the .material format but instead specifies the rendering and routing of full-screen render targets.

```
compositor "Night Vision" {
  technique {
    // Temporary texture(s)
    texture rt0 target_width target_height PF_A8R8G8B8
    target rt0 {
      // Render output from previous compositor
      // or original scene
      input previous
    }
    target_output {
      input none
      // Draw a fullscreen quad..
      pass render_quad {
        // ..using the night vision shader
        material Ogre/Compositor/NightVision
        input 0 rt0
} } } }
```

Listing 9: A simple OGRE compositor effect

Using this format it is possible to describe simple effects in a more concise way compared to X3D, while also allowing complex effects like pre-pending an invert effect to "Night Vision" (figure 2b) or even implementing deferred shading.

To enable compositing effects in X3D we added a new MFString field *compositors* to the Viewpoint node which allows specifying a compositor chain for that specific View. (see Listing 10)

4 PROPOSED X3D EXTENSIONS

Based on the discussion in the preceding sections we now propose two conceptual extensions to X3D that allow implementing the respective OGRE concepts directly instead of merely referencing them.

The first extension is the explicit notion of compositing by introducing a Compositing System inside X3D. The second extension is the definition of user defined appearances.

4.1 Compositing System

Following the OGRE notion of a compositor, we propose adding the following X3D nodes to allow the definition of a compositing effect directly inside a X3D file:

- *Compositor* for defining a named Compositor and the according scope. For defining the intermediate render layers, we use the RenderTexture extension.
- *CompositorPass* for explicitly stating the rendering order of RenderTextures. For specifying the shader and referencing input textures, we can use the Appearance node without modifications.
- *CompositorOutput* for explicitly marking the sink of a compositor graph. While one could use a special target on a CompositorPass, this makes automated error checking easier.

```
<Compositor DEF="GaussBlur">
  <RenderedTexture DEF="rt0" />
  <RenderedTexture DEF="rt1" />
  <CompositorPass target="rt0" input="none" render="SCENE"/>
  <CompositorPass target="rt1" input="none" render="QUAD">
    <Appearance>
        <ComposedShader USE="BlurVertical" />
        <RenderedTexture USE="rt0" />
    </Appearance>
  </CompositorPass>
  <CompositorOutput input="none" render="QUAD">
    <Appearance>
        <ComposedShader USE="BlurHorizontal" />
        <RenderedTexture USE="rt1" />
    </Appearance>
  </CompositorOutput>
</Compositor>
<Viewpoint compositors="GaussBlur" />
```

Listing 10: Sample usage of proposed Compositor node

Listing 10 shows a sample usage of the above nodes. The implemented effect is a separated Gaussian Blur filter which requires two render passes to be executed in the correct order.

4.2 User defined Appearance

Bringing user defined Appearances to X3D eases using specialized rendering techniques and allows bringing together the proposed Material extensions (Schwenk et al. 2012) (Sturm et al. 2016) using an unified concept.

```
<ComposedShader DEF="PBS">
[...]
</ComposedShader>
<CustomAppearance type="PBS">
  <!-- ComposedShader holds the type information -->
  <field name="roughnessFactor" value="0.4" />
  <ImageTexture url="albedo.png"
                containerField="albedoMap" />
</CustomAppearance>
```

Listing 11: User defined Appearance in X3D

To this end we propose the new CustomApperance node that references a named ComposedShader and can be used instead of the

classical Appearance node. The CustomApperance node contains any number of Texture and field nodes that are forwarded to the referenced shader.

5 CONCLUSION & FUTURE WORK

We have connected X3D to OGRE in a bidirectional manner allowing X3D scenes to be loaded by OGRE as well as using OGRE resources in X3D scenes. By comparing both on a conceptual level we could identify shortcomings in X3D and propose extensions to overcome those.

However we only extended X3D on a coarse level; one could improve the existing X3D concepts by comparing the implementations in detail. For instance one could improve the Geometry representation in X3D by allowing interleaved storage of vertex attributes and explicit buffer sharing by a notion of submeshes.

Furthermore our implementation, while already runnable on the web, only offers a custom C++ API. To allow using x3ogre as alternative to three.js or X3DOM, a SAI like API must be exported to JavaScript.

Finally it should be evaluated in how far the identified shortcomings also apply to the glTF format.

The implementation presented in this work is available open source at https://github.com/paroj/x3ogre.

REFERENCES

Johannes Behr, Peter Eschler, Yvonne Jung, and Michael Zöllner. 2009. X3DOM: a DOM-based HTML5/X3D integration model. In *Proceedings of the 14th International Conference on 3D Web Technology*. ACM, 127–135.

Johannes Behr, Yvonne Jung, Tobias Franke, and Timo Sturm. 2012. Using images and explicit binary container for efficient and incremental delivery of declarative 3D scenes on the web. In *Proceedings of the 17th international conference on 3D web technology*. ACM, 17–25.

James F. Blinn. 1977. Models of Light Reflection for Computer Synthesized Pictures. In *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '77)*. ACM, 192–198.

Robert L. Cook and Kenneth E. Torrance. 1981. A Reflectance Model for Computer Graphics. In *Proceedings of the 8th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '81)*. ACM, 307–316.

Leonard Daly and Don Brutzman. 2007. X3D: Extensible 3D graphics standard [standards in a nutshell]. *IEEE Signal Processing Magazine* 24, 6 (2007), 130–135.

Fraunhofer IGD. 2016. InstantReality — Avalon Rendering Engine. (2016). http://www.instantreality.org/story/what-is-it/.

Shawn Hargreaves. 2005. Generating shaders from HLSL fragments. *ShaderX3: advanced rendering with DirectX and OpenGL* (2005), 555–568.

ISO/IEC 19775-1. 2012. Extensible 3D (X3D) — Part 1: Architecture and base components. (2012).

Khronos. 2006. ARB_texture_env_combine. (2006). https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_texture_env_combine.txt.

Khronos. 2016. OpenGL ES 3.0 Specification. (2016). https://www.khronos.org/registry/OpenGL/index_es.php.

Khronos. 2017. WebGL 2.0 Specification. (2017). https://www.khronos.org/registry/webgl/specs/latest/.

Max Limper, Maik Thöner, Johannes Behr, and Dieter W Fellner. 2014. SRC-a streamable format for generalized web-based 3D data transmission. In *Proceedings of the 19th International ACM Conference on 3D Web Technologies*. ACM, 35–43.

Fabrice Robinet and P Cozzi. 2013. Gltf—The Runtime Asset Format for WebGL, OpenGL ES, and OpenGL. (2013).

Karsten Schwenk, Yvonne Jung, Johannes Behr, and Dieter W Fellner. 2010. A modern declarative surface shader for X3D. In *Proceedings of the 15th International Conference on Web 3D Technology*. ACM, 7–16.

Karsten Schwenk, Yvonne Jung, Gerrit Voß, Timo Sturm, and Johannes Behr. 2012. CommonSurfaceShader revisited: improvements and experiences. In *Proceedings of the 17th international conference on 3D web technology*. ACM, 93–96.

Steven Streeting. 2012. OGRE3D — Manual. (2012). http://www.ogre3d.org/docs/manual/.

Timo Sturm, Miguel Sousa, Maik Thöner, and Max Limper. 2016. A unified GLTF/X3D extension to bring physically-based rendering to the web. In *Proceedings of the 21st International Conference on Web3D Technology*. ACM, 117–125.

Alon Zakai. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 301–312.