3DRepo4Unity: Dynamic Loading of Version Controlled 3D Assets into the Unity Game Engine

Sebastian Friston*
University College London
London, UK

Carmen Fan
3D Repo Ltd
London, UK

Jozef Doboš
3D Repo Ltd
London, UK

Timothy Scully
3D Repo Ltd
London, UK

Anthony Steed
University College London
London, UK

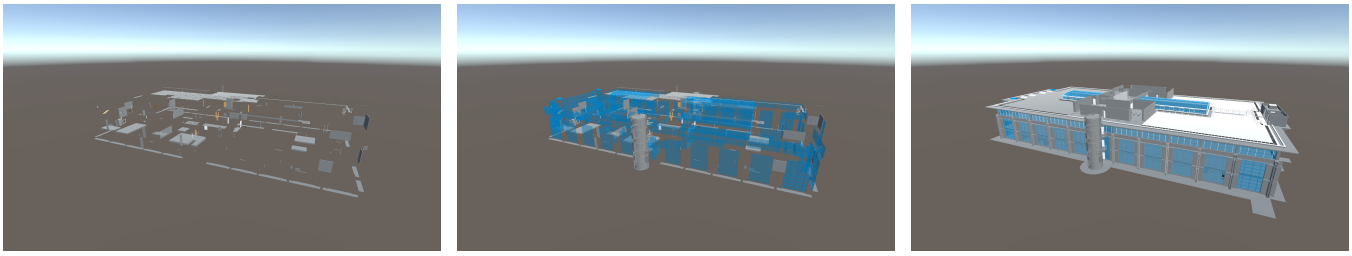

Figure 1: Test scene used in our measurements being dynamically loaded from a remote version controlled repository using a newly developed 3DRepo4Unity library. The building consists of 13k components and 4m vertices. Model courtesy of Skanska.

ABSTRACT

In recent years, Unity has become a popular platform for the development of a broad range of visualization and VR applications. This is due to its ease of use, cross-platform compatibility and accessibility to independent developers. Despite such applications being cross-platform, their assets are generally bundled with executables, or streamed at runtime in a highly optimised, proprietary format. In this paper, we present a novel system for dynamically populating a Unity environment at runtime using open Web3D standards. Our system generates dynamic resources at runtime from a remote 3D Repo repository. This enables us to build a viewer which can easily visualize X3D-based revisions from a version controlled database in the cloud without any compile-time knowledge of the assets. We motivate the work and introduce the high-level architecture of our solution. We describe our new dynamic transcoding library with an emphasis on scalability and 3D rendering. We then perform a comparative evaluation between 3drepo.io, a state of the art X3DOM based renderer, and the new 3DRepo4Unity library on web browser platforms. Finally, we present a number of different applications that demonstrate the practicality of our chosen approach. By building on previous Web3D functionality and standards, our hope is to stimulate further discussion around and research into web formats that would enable incremental loading on other platforms.

*sebastian.friston.12@ucl.ac.uk

Web3D '17, Brisbane, QLD, Australia

978-1-4503-4955-0/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3055624.3075941

CCS CONCEPTS

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

KEYWORDS

Unity, 3D assets, 3D Repo, MongoDB

ACM Reference format:
Sebastian Friston, Carmen Fan, Jozef Doboš, Timothy Scully, and Anthony Steed. 2017. 3DRepo4Unity: Dynamic Loading of Version Controlled 3D Assets into the Unity Game Engine. In *Proceedings of Web3D '17, Brisbane, QLD, Australia, June 05-07, 2017,* 9 pages.
DOI: http://dx.doi.org/10.1145/3055624.3075941

1 INTRODUCTION

Currently, many virtual reality (VR) and visualization applications are built on top of the Unity game engine. However, loading 3D assets at runtime, especially over the Internet, remains a challenge. Unlike dedicated WebGL-based environments on web browsers, Unity does not support streaming of common interchange data formats. Rather, streaming assets at runtime is supported through AssetBundles[1]. This effectively defines an opaque, proprietary format to deliver the assets in. The time it takes to produce these asset bundle files means in practice they must be pre-computed.

In the architecture, engineering and construction (AEC) industry, frequent design changes, coordination meetings and important presentations to non-experts are typical. Thus, quick and reliable visualization of large, up-to-date 3D projects remains a challenge.

[1]https://docs.unity3d.com/Manual/AssetBundlesIntro.html

Such issues are compounded by the fact that design teams—and the 3D assets—are often distributed across different time zones, organisations and platforms. To meet these challenges, some organisations have explored leveraging commercial game engines[2]. These engines have been developed for many years, incorporating lessons and features from many different projects. This is especially true of Unity, which provides many beneficial features and optimizations that are yet to appear in pure WebGL frameworks such as Three.js (Cabello 2010), X3DOM (Behr et al. 2009) or XML3D (Sons et al. 2010).

Due to its large user base and ease of deployment, e.g. support for native distributions including custom drivers for non-consumer hardware, Unity was chosen as our test-bed. Previously, Unity supported browser-based applications through a dedicated plug-in known as the Unity Web Player[3]. However, as of version 5.4, Unity decommissioned its Web Player in favour of indirect compilation into JavaScript assembly (Unity Technologies 2016b). This facilitates a more seamless and user friendly experience when distributing Unity applications across the web. Due to the need for cross-language compilation steps, not all system and processing libraries are supported in WebGL.

Overall, our aim is to democratize the creation and deployment of 3D visualizations for users who are skilled in 3D content generation, but who do not necessarily have the knowledge or time to develop bespoke Unity applications. Such users not only include engineers working with Computer Aided Design (CAD) and Building Information Modelling (BIM), but also independent game developers and hobbyists alike. By delivering a web-enabled platform, these professionals would be able to share their 3D scenes with colleagues and collaborators without needing to transfer large amounts of information in proprietary formats over traditional, manually-intensive distribution channels, such as file-shares.

In this paper, we set out to design, develop and test a novel way of populating a running Unity application with assets that are dynamically loaded from a representational state transfer (REST) application programming interface (API) over the Internet as shown in Fig. 1. Then, we compare the loading performance with an existing visualization client based on declarative X3DOM using a set of experimental measurements. Even if the Unity application is compiled into a highly optimized version of JavaScript that should outperform unoptimized X3DOM, Unity comes with many libraries and additional processing overheads that may affect the loading and display times of equivalent assets, as discussed in §8. Although our current implementation is based on X3D scene descriptors, in the future it can be easily expanded to support emerging formats such as GL Transmission Format (glTF) (Khronos Group 2015) and Draco (Google 2017).

Contributions. Our contributions can be summarized as follows:

(1) Design and implementation of a novel Web3D-based asset loader on top of the Unity game engine.
(2) Evaluation and comparison of the asset loading speed between 3drepo.io and 3DRepo4Unity clients.
(3) Validation of the proposed solution in a real-world scenario.

[2]http://www.aecmag.com/59-features/1166-virtual-reality-for-architecture\-a-beginner-s-guide

[3]https://unity3d.com/webplayer

2 RELATED WORK

Many systems support dynamic loading of web-based assets at runtime. Early VR platforms such as Distributed Interactive Virtual Environment (DIVE) (Carlsson and Hagsand 1993) and Model, Architecture and System for Spatial Interaction in Virtual Environments (MASSIVE) (Greenhalgh and Benford 1995) delivered assets online to support cooperation between geographically dispersed users. However, they were based on decentralized peer-to-peer connections.

VRML97 (Web3D Consortium 1997) and X3D (Web3D Consortium 2013) specify sub-scenes with Uniform Resource Locators (URLs), splitting data across multiple files. This reduces initialization time compared to hard-coding assets within the Hypertext Mark-up Language (HTML) as done in early declarative 3D on the web, c.f. (Behr et al. 2009; Sons et al. 2010). In such cases the entire scene had to be downloaded and parsed first, blocking the main GUI thread and causing the browser to become unresponsive.

With the introduction of WebGL, such problems inspired a proliferation of 3D REST APIs and externalized file formats. For example, AMD sponsored the *rest3d* initiative (Parisi and Arnaud 2011) that proposed a unified REST interface for all 3D assets online. Schiefer et al. (2010) introduced a REST service integration for OpenSG and Doboš et al. (2013) defined a REST API for XML3D. Nevertheless, large scenes still consist of hundreds of thousands of components. Although web browsers can fetch multiple XMLHttpRequest (XHR) requests in parallel, they only support several thousand at a time. This led to the introduction of new file formats such Shape Resource Container (SRC) (Limper et al. 2014) and Blast (Sutter et al. 2014). To cope with large models, bounding interval hierarchies (BIH) were utilized client-side for visibility culling (Stein et al. 2014) and Multi-Part X3DOM nodes for mesh batching (Scully et al. 2015). Based on such advances, (Mouton et al. 2014) demonstrated a scalable system for model-driven web services architecture targeting CAD online. Such efforts culminated in the introduction of glTF (Khronos Group 2015) with various extensions such as binary glTF (Cozzi et al. 2016) for buffer views as well as streaming (Scully et al. 2016). Other systems also interlinked various web assets, e.g. (Olbrich 2016) for X3DOM, (Schilling et al. 2016) for CityGML streaming to CesiumJS and (Lu et al. 2016) for crowd-sourcing of city models.

In contrast, large-scale virtual worlds such as SecondLife (Rymaszewski 2007) and its open source C# derivative OpenSim host multiple worlds in a grid composed of hyperlinked regions. This is based around a custom-built protocol for dynamic exchange of assets over the Internet. Since these predate the most recent Web3D formats, they support mainly sculpted primitives and bitmaps, but also COLLADA files. Similarly, Open Wonderland built in Java provides a shared online 3D space where users interacted with dynamically loaded COLLADA files, as well as 2D windowing systems including presentations and web browsers from within 3D space.

Recently, systems such as Google Earth VR and Autodesk Live have begun to support rendering web-based 3D assets in VR. These systems are proprietary and closed-source, though there are open interoperability tools. For example RawKee is an open source add-in to Autodesk Maya that exports character animation and scripting into X3D.

3 ARCHITECTURE OVERVIEW

Our solution is built on top of the Unity game engine. Leveraging game engines allows developers to quickly get started on platforms with all the basic features required to display a virtual world, massively reducing non-recurring engineering costs. Further, a team will have access to the latest developments in computer graphics that would be beyond the abilities or budget of a smaller group working alone. Unity was chosen specifically because its licensing model has no up-front costs, is a popular choice for lower-end hardware, and its popularity provides ready access to a community.

However, the requirements overlap between video games and construction visualization is not perfect. Developers leveraging game engines must be prepared to work around design decisions that in some cases are actively detrimental to visualization applications. For example, the assets forming a game world are mostly static and highly constrained ahead of time. This allows the engine to make pre-computations and assumptions for the sake of performance. It also means support for loading arbitrary assets at runtime is minimal, if it exists at all. This is true of Unity, which has an additional complication in that its API is not thread-safe making asynchronous loading extra challenging. Thus, the architecture of the proposed solution consists of three independent systems: i) 3D Repo version controlled Web3D repository, ii) Unity game engine, and iii) 3DRepo4Unity stand-alone C# library.

3.1 3D Repo

3D Repo (Doboš and Steed 2012), a domain-specific open source version control system, has evolved from an XML3D based (Sons et al. 2010) visualization to a AngularJS and X3DOM (Behr et al. 2009) based commercial platform, *3drepo.io* (Scully et al. 2015). With the advent of glTF, 3D Repo added experimental support for web-based streaming and direct GPU memory manipulation (Scully et al. 2016) in JavaScript and WebGL. The core of the platform consists of several interconnected systems. A C++ processing back-end loads, decomposes and optimizes industrial 3D models. These are stored in a polymorphic Binary JSON (BSON) collection within a NoSQL database in 3D Repo's internal scene graph format. This resides alongside a revision history and various highly optimized Web3D data representations. A Node.js web server provides dynamic web pages and exposes a REST API. The API can be queried for various resources including user profiles, projects, histories, revisions, scene graphs, externalized meshes & textures, etc., all in multiple formats and encodings. Most of these are generated on demand, depending on the requested data representation (XML, JSON, SRC, glTF, etc.). However, such responses can be cached server-side. This is typically beneficial as data storage is, in general, cheaper than CPU processing time. Although 3D Repo does not provide direct 3D editing functionality, users can upload revisions and post comments within them. The comments themselves follow the BIM Collaboration Format (BCF) standard (buildingSMART 2016) by buildingSmart.

3.2 Unity

Unity is a well known game development framework. It consists of a real-time rendering engine with support for systems such as audio, physics simulation, user input and networking. Developers build applications using the Unity Editor, in which they interact directly with the Unity engine's systems using a graphical user interface (GUI), rather than building a scene graph in code. The editor then builds distributables targeted at a number of platforms. Unity itself is closed source, but its licensing is friendly to small and medium enterprises (SMEs) as the revenue sharing model allows full access to almost all its features with no upfront costs.

A Unity application is based around a scene graph, formed of GameObjects. As explained in Section 3.2, all application functionality exists as an element of one or more GameObjects. The Editor supports many common 3D interchange formats. However, these are converted at design time to native types, before they can interoperate with Unity's systems. Similarly, Unity supports optimizations such as static batching, but this requires the assets and scene graph to be defined at compile time. While it is possible to create native resources at runtime, not all the importer functionality is present.

Programming paradigm. Unity utilizes the component based model for engine design. It contains multiple systems, controlled by attaching *Components* to objects in the scene graph. Components are instances of managed classes that form the Unity API. For instance, a Component may integrate with the rendering pipeline to draw geometry or play audio. The engine does not have a comprehensive central static API. However, some low-level functionality is accessible via specific interfaces. For example, it is possible to make calls analogous to a subset of OpenGL that operate below the rendering pipeline. Generally, each system is treated atomically and as a black-box, with behaviours controlled through the managed Components only.

Thus, a developer writes the behaviour of an application by creating their own Components. These are instances of Unity's *MonoBehaviour* class, which is a generic base class capable of integrating into the Unity scene graph on a GameObject. The class includes methods to be overridden which are essentially callbacks. Within these methods, the developer may interact with the programmatic API of other Components, or execute any other code as desired. Developers write these Script Components in C# or JavaScript, with the Unity Editor automatically invoking the C# compiler and integrating the binaries into an application with a common runtime.

3.3 3DRepo4Unity Library

While Unity does support runtime loading of assets, the implementation is not flexible enough for our requirements. The dedicated API is only designed to work with bundles of assets delivered in a proprietary format (Unity Technologies 2016c). The importers for typical interchange formats are only available within the editor, not at runtime. For example, textures can be loaded from byte arrays but meshes must be constructed manually.

Over time, 3D Repo has evolved to deliver scene content in X3D supported formats, thus the new Unity based client must be able to interpret the X3D scene description. Although Unity supports modifying the scene graph dynamically, its programming model is single threaded. Unity does have the concept of co-routines, but these are designed to distribute execution of a single function call across multiple frames, rather than multiple threads (Unity Technologies 2016a). Unlike heavily optimized game models, the environments loaded by 3D Repo are large and unoptimized; with assets

delivered between potentially thousands of Hypertext Transfer Protocol (HTTP) API calls across a wide-area network with varying latency. Hence, support for multi-threaded loading is necessary to ensure the user interface remains responsive.

Responsibilities. In response to these challenges, we designed our solution to be as decoupled from any Unity-specific design-decisions and data types as possible. Furthermore, we aimed to compartmentalise the main responsibilities to reduce friction during future maintenance. These responsibilities are:

(1) API translation and abstraction of the server by the Web Client
(2) Multi-threaded scene data import by the Importer
(3) Application-agnostic scene representation with the Portable Scene Graph
(4) Loading the scene data into Unity with an `ISceneGraphTranslator` implementation

We decided to build a .NET library, separate from Unity itself. This allows the library to be reused outside of Unity in contexts such as automated test suites, as well as in other applications.

The interface to Unity itself is deliberately kept narrow. The library presents a C# API suitable for use by Unity Script Components. The API consists of a set of methods presented by the `_3dRepoEngine` object, and an internal scene graph. `_3dRepoEngine`'s methods are high level, suitable for binding directly to a GUI permitting a user to control the loading process. The library handles all communication with the server by requesting and parsing scene data into the internal platform agnostic scene graph. Through a callback mechanism, Unity iteratively synchronizes this internal graph with its own as explained in §6. The callback is made each frame, during which any new data is incorporated into the native Unity scene. The graph is designed to be as simple as possible so that converting it to a native representation requires minimal overhead and all application-specific code can remain within Unity. A data flow diagram is shown in Fig. 2, showing how the solution's components can be grouped at a high level into each responsibility.

4 WEB CLIENT

The web client consists of `RepoController` and its dependencies, as illustrated in Fig. 2. These classes together translate the 3D Repo REST API into a C# API that is used throughout the remainder of the solution. All interaction with the server takes place through a `RepoController` instance. This in turn uses a number of resources to exchange HTTP requests, maintain a session and a local cache.

`RepoController` is a wrapper around the 3D Repo REST API. It emits HTTP requests based on method calls, and parses the JavaScript Object Notation (JSON) responses into equivalent native objects. That is, it is a managed wrapper around the API published by 3D Repo. Higher-level functionality, such as navigation through revisions, is performed by `_3dRepoEngine` using `RepoController`, which is stateless.

The 3D Repo API uses cookies to maintain a session. These are managed by `HTTPClient`. `HTTPClient` is what synthesizes the HTTP requests and interoperates with the operating system's network stack. It has no application-specific functionality, other than keeping the 3D Repo cookie. It was created only because the target version of .NET did not have a sufficiently capable client built in.

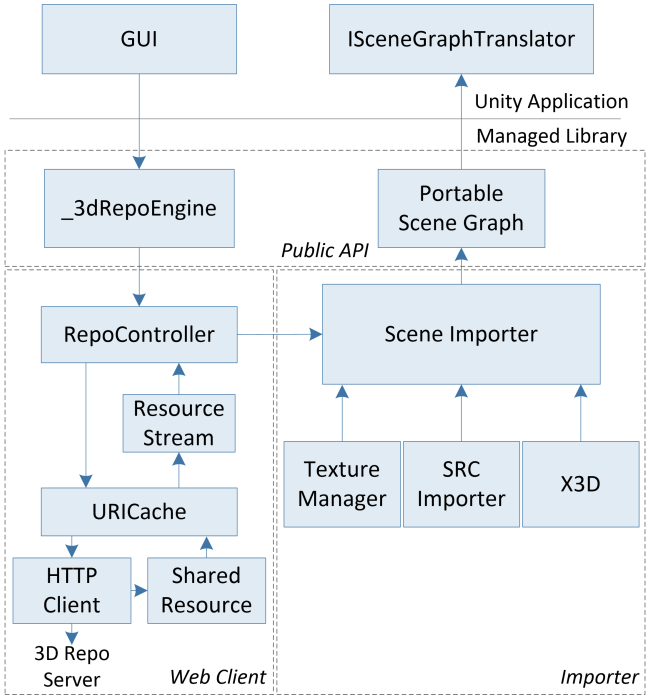

Figure 2: Simplified Data Flow Diagram of 3DRepo4Unity.

Resource Management. `RepoController` does not parse any data that is not part of the 3D Repo REST API specification. All scene data is passed as raw byte-streams to `SceneImporter` as `ResourceStreams`. 3D Repo is designed to deliver scenes in X3D which is effectively an Extensible Markup Language (XML) for describing fully interactive 3D scenes. X3D uses Uniform Resource Identifiers (URIs) to identify resources and sub-trees, and resources may be shared between multiple nodes. To emulate the natural behaviour of a web browser, HTTP requests are routed through an instance of `URICache` to avoid downloading the same resources multiple times. `URICache` caches server responses as instances of `SharedResource`. `SharedResource` instances receive server responses, downloading large assets on separate threads, and provide a thread-safe API to the streams. `ResourceStream` instances wrap this API in a stream-like object for use by native .NET classes. As the REST API is used to make requests for changing data, such as revision and branch information, `RepoController` only routes a subset of requests through `URICache`.

High-level API. Unity interacts with the library by instantiating `_3dRepoEngine`. `_3dRepoEngine` has methods for logging in and out of the server, and requesting revisions. The intent is for these methods to be high-level enough that they could be bound directly to GUI controls and by keeping them separate from `RepoController`, new conceptual uses of the API can be easily added to the engine as it evolves. When a call to one of the high level methods is made, `_3dRepoEngine` will invoke the appropriate REST calls in `RepoController` and route the responses to an instance of `SceneImporter`. `SceneImporter` will parse the responses, load the scene data and deliver it to Unity as described in §5 and §6.

5 PORTABLE SCENE GRAPH

Unity receives scene data through our portable scene graph, which consists of a hierarchy of `PortableSceneNodes`. These nodes may have geometry and materials attached to them, just like the native Unity scene graph. The portable scene graph classes are highly rigid and simplistic, encapsulating only the bare functionality used by 3D Repo scenes. Members are strongly typed with implicit semantics to make the conversion to native equivalents straightforward. These types will change as 3D Repo evolves (see §6.1 and 6.2).

The portable scene graph addresses the challenges of asynchronous loading in Unity. It is designed so that branches can be constructed in parallel and then merged together. Node methods are inherently thread-safe, with each branch having a context object, i.e. a mutex, that locks critical sections when a branch is updated. The context object can also be used to lock the entire branch externally, such as when single-threaded applications such as Unity need to traverse the entire graph. When a branch is grafted onto another, it adopts the parent's context and from then on is seen as one branch. This approach decouples the thread-management of the scene graph from that of the scene importer, simplifying both.

Scene loader. When instructed by the host, `_3dRepoEngine` will request the X3D scene for a revision by issuing the appropriate calls to `RepoController`, in accordance with the published 3D Repo API. 3D Repo delivers X3D with additional third-party formats such as SRC (Limper et al. 2014), and typical image texture formats. The raw response data is routed to the `SceneImporter`. This class and its dependencies parse the raw data into portable representations of basic assets (geometry and textures) that can be loaded into Unity, or any other real-time engine. Once the X3D scene is parsed, `SceneImporter` traverses the generated scene Document Object Model (DOM) and translates it into our portable scene graph. New scene graph nodes are created as new X3D nodes are encountered, and the recursive traversal methods take instances of both ensuring that nodes are added in the correct place. It is also in `SceneImporter` where the semantics implied by an X3D type are used to translate the scene elements into concrete types.

Multi-threaded support. The actual `SceneImporter` used by the library is an asynchronous subclass. Simple conditionals decide if a given DOM node should be imported in its own thread. For example, Inline nodes, which contain sub-scenes, or nodes referencing geometry, are imported in new threads. This is because there may be a non-trivial delay while dependencies for a node are downloaded, during which time other nodes could be processed. It also allows for simultaneous downloads of resources, that further reduces perceived latency. If an import is aborted or suffers an error, the `AsyncSceneImporter` aborts its workers before destructing, and these aborts are propagated, recursively terminating all threads until only the root remains. To retrieve the imported scene data, the application needs to traverse the portable scene graph. The mechanism to do so is for an application to provide a callback in the form of an `ISceneGraphTranslator` instance (see §6). The application assigns this as a member of the `_3DRepoEngine`, then calls an update request method that blocks until pending updates to main graph are made. The graph will not be locked while the geometry is downloaded, since `AsyncSceneImporter` would construct the nodes outside of the main graph and only then append them once complete. When permissible, the update method locks the graph and a method of the `ISceneGraphTranslator` interface is called. This loads the changes in the graph into the host application. The update request call is simply a helper method. It retrieves any worker exceptions, the state of the import, and handles locking of the graph. It directly calls the `ISceneGraphTranslator` update method, however, and so both this and the update method execute in the same thread. Finally, `ISceneGraphTranslator` is responsible for translating portable scene graph into a native one.

6 UNITY IMPORT

As multiple `AsyncSceneImporters` run in parallel, and independently download resources of varying sizes, the order in which different branches of the DOM/scene are completed is arbitrary. For example, a scene root node with of a set of Inline child nodes could be traversed long before the nodes themselves complete in separate threads, as they may have "heavy" geometry to download. Thus, each `AsyncSceneImporter` builds a portable scene branch independently. Once a DOM has been exhausted, the branch is grafted into the main portable scene graph referenced by the initial `AsyncSceneImporter` instance.

The instance of `ISceneGraphTranslator` is invoked by Unity every frame through the update request method, which runs in the main Unity thread. On each invocation, the update method traverses the Unity scene graph and the portable scene graph in lock-step. For each level of the graph, it attempts to match any portable graph child nodes to existing Unity GameObjects, the native Unity scene graph node type. The order of the child nodes may change between the graphs, so matching is performed based on node names. An alternative would be to give each Unity node a component storing a unique identifier (UID). If a corresponding Unity node cannot be found, a new one is created. Regardless of whether or not matching nodes have been found, the traversal continues for all their children. This is because new nodes may be added at any depth and so the entire graph must be traversed to ensure all changes are incorporated. To make this process more efficient, the portable graph is furnished with flags allowing both individual nodes and the graph as a whole to signal in what ways, if any, it has changed since the last call. These are cleared by the `ISceneGraphTranslator` implementation. It is within the `ISceneGraphTranslator` object as well that native Unity Mesh and Material types are created and added to the native scene. Nevertheless, it is important to note that this lock-step traversal happens only while the assets are being loaded over the Internet so it has zero effect on the rendering performance thereafter.

6.1 Geometry Import

Although many X3D nodes describe visible geometry, 3D Repo utilizes only the `ExternalGeometry` node, which references an SRC mesh. This is mainly to avoid large amounts of data being stored directly within the scene graph definition which would significantly increase the perceived lag when parsing an X3DOM page. SRC was designed for transmission of indexed face data, primarily as a delivery rather than a data interchange format. It interoperates

closely with WebGL so that its data can be directly loaded onto the graphics processing unit (GPU).

Despite the low-level arrangements of SRC being suitable for WebGL and OpenGL ES, it becomes more complex with Unity's more abstract API. While Unity does support runtime creation of meshes, it constructs GPU buffers with different layouts based on the available vertex channels and shader capabilities. To import geometry, `SceneImporter` passes SRC formatted streams to the `SRCImporter` which in turn parses them into an equivalent managed type. Then, using the information in the SRC header, the interleaved mesh data is extracted into individual but contiguous channels. Finally, the semantic information from the header is used to assign these to specific members of the portable mesh type.

3D Repo also utilises the X3DOM MultiPart (Scully et al. 2016) node. Its purpose is to support submeshes as a form of mesh batching. In addition to the actual mesh data, defined by an `ExternalGeometry` node or similar, an array is provided that assigns each primitive to a particular submesh, and materials are assigned on a submesh basis. In the X3DOM runtime, these indices are used to lookup material properties in the fragment shader. Unity has a native support for submeshes, so the mesh importer uses the array to group indices into submeshes after the geometry has been transcoded. All portable mesh nodes have at least one 'submesh', which contains faces. If created within a MultiPart node, these are subsequently split into a set of submeshes before the portable mesh is returned.

6.2 Material Import

X3D has a number of material definition options, including the ability to specify arbitrary shaders. Again, we support only the subset required by 3D Repo. 3D Repo uses the `X3DMaterialNode` exclusively. This node explicitly defines a set of basic material properties such as diffuse and emissive colors, and texture maps. These properties are parsed directly into the portable material type by `SceneImporter`. Within Unity, a small subset of shaders are used for all materials, chosen depending on whether the material should support features such as transparency or texture mapping. All material node parameters have analogues in these shaders. Still, translating materials between runtimes is typically a significant challenge. Even when the same lighting models are used, there are usually differences in the implementation. This is because individual parameters cannot be used directly and some sort of conversion heuristic needs to be applied. The use cases of 3D Repo are typically tolerant of simplistic materials and lighting however, making the subset of properties used in 3DRepo4Unity tenable. Fortunately, constraining the server to `X3DMaterialNode` makes material import quite simple. The exception is the handling of textures. Though `URICache` allows reuse of locally cached byte arrays, if these were to be passed to an image parser multiple times, it would still result in multiple copies of the same image in memory. Thus, we introduce a `TextureManager` instance, which caches images based on URI in `SceneImporter`. `X3DMaterialNode` is part of the X3D specification, so unlike geometry, it is handled by the `SceneImporter`. By reusing the same parsed image references, the `ISceneGraphTranslator` can similarly reuse a dictionary of references to re-use the same native images once loaded in Unity.

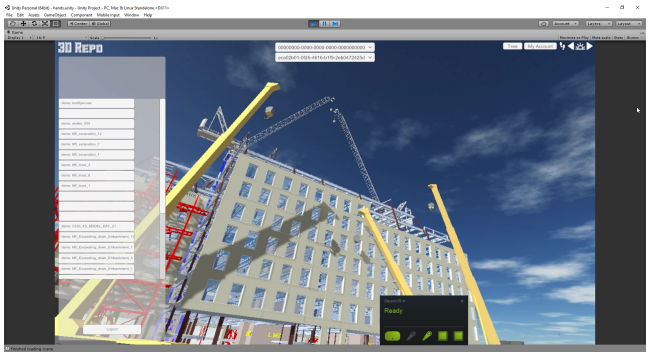

Figure 3: 3D assets are dynamically loaded into a running Unity application over the Internet. Upon loging in using the same credentials as in 3drepo.io, the system lists all available projects (left) and enables the users to hot swap them on the fly. Branches and revisions can be selected using drop down menus (top middle).

7 USER INTERFACE AND DEPLOYMENT

Unity provides native support for building 2D GUIs. Widgets are represented by GameObjects in the main Unity scene graph and have components that interoperate with Unity's systems to automatically size and place themselves. Navigation throughout the 3D environment is done with traditional video game paradigms. The 2D GUI in our case has the responsibilities of authentication and settings management, as well as navigating a project's revision hierarchy. The GUI, shown in Fig. 3, mimics the 3D Repo browser-based client. Unity has a component based programming model, described in §3.2, so that various elements of the Unity application are independent by nature. Yet, to make an intuitive UI, a notion of a state is required. For example, if the user is not logged in, no other widgets should be visible. Thus we introduce `UIManager` as a component which shows or hides individual aspects of the UI depending on the state of the system, and the user's preferences.

Environmental controls. 3D Repo revisions do not contain detailed ambient environment geometry or lighting. Further, descriptions of such environments are for now very application-specific. For example, some runtimes may support a limited number of basic lights, while others use high-dynamic-range imaging (HDRI) environment maps. Ambient environments, therefore, are handled entirely on the Unity side. Its shader system supports environment maps, basic lights, and a number of options for dynamic global illumination. These can be specified dynamically, just like scene geometry, so this functionality could be extended in the future. Builds could also be made with customer-specific environments. A number of basic environments are built into the solution. The revisions do, however, support preconfigured viewpoints, and these are represented within the portable scene graph. When encountered, the scene translator will alter the camera transform to mirror them.

7.1 WebGL Compilation

Since Unity 5.4, the Webplayer has been phased out in favour of native WebGL support. This is achieved by translating Unity game

code into C++ using *IL2CPP* which is then further compiled into JavaScript using *emscripten* (Unity Technologies 2016b). Whilst creating a Unity WebGL build is as simple as changing the target platform on the SDK, there are several limitations to consider.

Libraries support. Due to the restrictions of JavaScript and web browsers, not all features supported within Unity are supported in WebGL. For instance, some external JSON libraries and the .NET XML Deserializer are not supported. However, the most critical ones are System.Threading and System.Network. Due to the lack of multi-threading, we can no longer support loading of scene data in parallel. Nevertheless, some level of asynchonrous processing can still be achieved due to reliance on JavaScript. In comparison to a desktop version, the HTTP client also requires a full rewrite to utilise UnityEngine.WWW instead of System.Net. This maps directly onto the XHR request with a JavaScript-like control flow using *UnityEngine.Coroutine* and *yield*. Thus, scene fetching functionality had to be adapted in a way where the work is only done when the data has been yielded, using callback functions. Note also that whilst C# supports function pointers originating from instanced objects, WebGL supports only static functions, like native C++. To further maintain a valid session, the application needs to provide a session cookie with every request. Due to web browser restrictions on the header, it is not possible to manually extract and insert this cookie on a XHR request in a WebGL build. Credential request `withCredentials` header is also not natively supported. As of Unity 5.5, the only way to achieve this is to utilize a plug-in script to add `withCredentials` into the header (Unity Technologies 2016d).

Memory allocation. Unity in WebGL manages its own memory by requesting a block from the browser's heap at the initialization stage. This is known as the *Unity Heap*. There, it handles all memory allocation inside the Unity Engine. The size of the heap is configured at compile time, thus the Unity WebGL consumes a constant amount of memory. To increase the memory usage, the build would have to be recompiled, with a maximum allowed size of 2039MB. This has proven to be a difficult restriction for our use case, as the sizes of BIM models could be either too small or too big. A naïve solution would utilize the biggest possible memory buffer, meaning clients with small models would still be consuming a large amount of memory while large models could run out before being initialized.

8 EVALUATION AND DISCUSSION

Fig. 1 shows our 3DRepo4Unity library loading assets from a remote location over time. The users are able to successfully log in, maintain a session and retrieve any projects that are already stored within the 3drepo.io web server as further shown in Fig. 3. To evaluate the usefulness of such an approach in day-to-day engineering, we have developed a VR application on top of this library for the purposes of health and safety induction (being demonstrated in Fig. 4). Many commercial engines, including Unity, have inbuilt support for popular VR headsets. This allows the same solution to be used for navigation on desktops and in VR through configuration of only a few settings. In our construction site office deployment, the project delivery coordinators who are skilled in 4D planning

Figure 4: Health & safety induction using 3DRepo4Unity at the M5 highway site office of Balfour Beatty and Vinci in the UK. Project video available online at https://www.youtube.com/watch?v=WGzwacVP66U

Figure 5: The same test 3D scene as shown in Fig. 1 visualized in 3drepo.io based on X3DOM. For the purposes of our experimental evaluation, all user interface features have been disabled to match the 3DRepo4Unity base performance.

using software packages such as Synchro and Autodesk Navisworks were able to upload their custom site-specific 3D models successfully, defining specific vantage points and exclusion zones throughout the existing 3drepo.io web interface. These assets were then loaded using the 3DRepo4Unity desktop VR application for training purposes.

Experimental evaluation. To compare the loading speed and performance of 3DRepo4Unity, we have performed an experimental evaluation on a medium sized 3D model as shown in Fig. 5. As a baseline, we used the existing 3drepo.io platform running X3DOM. Against this, we compared the 3DRepo4Unity compiled in WebGL mode as well as the off-the-shelf WebGL Unity with the model delivered in native Asset Bundle data format. Median values from five separate runs are reported in Fig. 6. These were calculated to reduce the influence of outliers. Each timing consists of the respective engine's initialization time, fetching of the 3D assets, and subsequent GPU loading via built-in timestamp printouts.

Unity has a longer initialization period as it requires the Engine and Heap to initialize. Note that whilst the graph shows the Asset

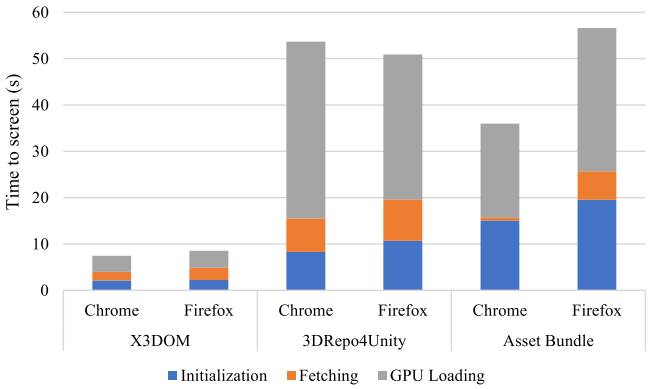

Figure 6: Median values over five runs of time to screen (in seconds) between 3DRepo4Unity and Asset Bundle across two popular web browsers. X3DOM timings act as baseline.

Bundle version taking longer to initialize than 3DRepo4Unity, the Asset Bundle version was compiled without stripping the engine, i.e. discarding libraries which are potentially not being utilized. The libraries within the engine that are required for the Asset Bundles to work are not known at compile time, thus if we were to strip down the engine, it might be missing critical functionalities for them to load correctly. This is something that can be improved upon in the near future. Consequently, the size of the Asset Bundle engine is significantly higher than that of 3DRepo4Unity. The 3D Model we used in this experiment has a comparable size in X3D Format vs Asset Bundle which uses LZ4 compression, thus we would envisage the fetching period to be similar in our timings. However, it is clear that 3DRepo4Unity, fetching the same files as X3DOM, took significant longer to perform the XHR requests. It is unclear why the difference exists especially without deeper understanding of how Unity translates the application code into JavaScript. Nevertheless, we suspect that one of the reasons may be due to the compiled code not utilizing the asynchronous nature of JavaScript, unlike X3DOM. Overall, our implementation is comparable in performance to the native Asset Bundle support which in itself is a very positive result. What is more, we found that Asset Bundles even on medium sized construction models take a long time to generate. In the particular case of building shown in Fig. 1, it has taken over five hours to generate. Consequently, we do not consider the small run time performance gain being worth the preprocessing requirements.

The results in Fig. 6 also show that X3DOM performs significantly better than Unity on WebGL. X3DOM, being a polyfill WebGL renderer, benefits from its ability to utilize the browsers' parallel loading functionality in full. What is more, it renders X3D natively, whereas Unity has to transcode X3D or even its own Asset Bundles into GameObjects at runtime, yielding a longer GPU propagation. Requiring nearly one minute from pressing a button to seeing the finished rendering on screen, such an approach would still need to be optimized further to be truly usable in real world.

Finally, it is important to note that 3DRepo4Unity consumes double the amount of memory in comparison to Asset Bundle version. This is expected as the implementation we chose requires an extra copy of the scene graph in the form of the *PortableSceneGraph*. As mentioned in §7.1, Unity WebGL builds are limited to $\tilde{2}$GB of memory which severely limits the size of models that can be rendered on the platform. Thus, in a commercial environment, we may have to sacrifice interoperability for performance. However, we also noticed that the Unity engine has good memory management; once the scene is loaded onto the GPU, the CPU memory consumed in generating the scene graph is garbage collected, allowing more CPU memory to be used to perform other functionalities. In contrast, X3DOM will permanently hold onto the CPU copy of the scene.

Advantages. In our implementation, the `SRCImporter` is separate to the scene importer. This is because the SRC format is independent of X3D, but also because it enables geometry importers to be added and extended more easily, especially once 3D Repo fully moves to another format such as glTF, c.f. (Scully et al. 2016). By first transcoding geometry data to a more generic representation, changes to the portable mesh type can be made more easily and conversion to this type remains independent of a particular geometry parser which is a plus. Although our current implementation supports only a small subset of the X3D specification, there is in theory no reason why it would not be possible to extend the library to support the entire X3D feature set. Special care would have to be taken when accommodating embedded actions, animations and scripts. This would enable designers and non-programmers to create interactive 3D scenes using ready-made Web3D tools. It would also provide greater interoperability with existing systems across the Internet. Nevertheless, for our current requirements in the AEC industry, our standard support for geometry, mesh batching and textures is all that is required. What is more, the current Unity WebGL implementation does not support built-in XML parsers which makes any X3D-based import functionality limited to non-browser deployments only (see §7.1 for further details). With the advent of glTF and its inclusion of a scene graph, it is also questionable whether full support for X3D would be necessary. However, as it stands, our 3DRepo4Unity library should be able to seamlessly load most X3DOM-based assets on desktops.

Limitations. The main limitation of Unity on the web is the overhead associated with initializing and running the required libraries in a web browser. In comparison to direct WebGL implementations of X3DOM, there is a significant delay when loading the assets remotely as well as when uploading to the GPU. In addition, Unity design decisions sometimes lead to conflicts between following the highly compartmentalised paradigm and achieving optimal performance. For example, to keep our solution flexible, the object highlighting behaviour in the browser is implemented in Unity as a second render pass using stencils, rather than a modification to the shader itself (which would require modifying all shaders, now and in the future). This decoupled design is in-keeping with Unity's programming model and is more flexible. For example, the highlighting behaviour can be altered on a camera by camera basis, by changing the attached component. This could be used to draw different highlights in VR than on the desktop, or perhaps none at all. With no dependencies in the primary materials and rendering pipeline, it is easier to upgrade to the latest Unity materials, and also integrate third-party extensions. However, it requires that the geometry must be drawn twice. Given the sizes of many 3D Repo revision scenes, this is not a trivial consideration.

9 CONCLUSIONS

In this paper we have devised, implemented and tested a novel method for dynamic loading of Web3D assets from a REST API over the Internet into the Unity game engine at runtime. Although Unity supports loading assets at runtime, our implementation is more flexible, platform agnostic and can be extended to other engines. This implementation was evaluated in an experiment whereby the state-of-the-art X3DOM-based 3drepo.io client was compared and contrasted with 3DRepo4Unity as well as native Asset Bundle implementations using WebGL. This showed that our implementation provides comparable performance while unleashing the power and capabilities of a fully-fledged game engine on the web. Our hope is that this and similar work will stimulate further development and support for open Web3D standards in popular game engines soon.

Future work. Soon, we plan to expand our implementation with support for glTF so that the 3D Repo SRC-based API can be phased out. Then, we will attempt to add streaming and memory management capabilities akin to (Scully et al. 2016) so that theoretically unlimited virtual environments can be traversed in real-time over the Internet despite limited server infrastructure. Furthermore, the move towards physically based rendering (PBR) on the web will be a significant benefit for the transfer of materials between runtimes, as the scope for interpreting individual parameters in PBR is by definition narrow. Unity has already moved their material system to PBR, and common formats for their material descriptions are already proposed for the web, c.f. (Sturm et al. 2016).

ACKNOWLEDGMENTS

This project has been funded by Innovate UK under the Infrastructure Systems grant No. 102813 for which we are grateful. 3D Repo open source initiative has been also kindly supported by EIT Digital and Digital Catapult in the UK.

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 (Web3D '09)*. ACM, New York, NY, USA, 127–135. DOI: https://doi.org/10.1145/1559764.1559784

buildingSMART. 2016. Web service specification for BIM Collaboration Format. (January 2016). https://github.com/BuildingSMART/BCF-API.

Ricardo Cabello. 2010. Three.js JavaScript 3D library. (2010). https://github.com/mrdoob/three.js/.

C. Carlsson and O. Hagsand. 1993. DIVE A Multi-user Virtual Reality System. In *Proceedings of the 1993 IEEE Virtual Reality Annual International Symposium (VRAIS '93)*. IEEE Computer Society, Washington, DC, USA, 394–400. DOI: https://doi.org/10.1109/VRAIS.1993.380753

Patrick Cozzi, Tom Fili, Kai Ninomiya, Max Limper, and Maik Thöner. 2016. KHR_binary_glTF. (2016). https://github.com/KhronosGroup/glTF/tree/master/extensions/Khronos/KHR_binary_glTF.

Jozef Doboš, Kristian Sons, Dmitri Rubinstein, Philipp Slusallek, and Anthony Steed. 2013. XML3DRepo: A REST API for Version Controlled 3D Assets on the Web. In *Proceedings of the 18th International Conference on 3D Web Technology (Web3D '13)*. ACM, New York, NY, USA, 47–55. DOI: https://doi.org/10.1145/2466533.2466537

Jozef Doboš and Anthony Steed. 2012. 3D Revision Control Framework. In *Proceedings of the 17th International Conference on 3D Web Technology (Web3D '12)*. ACM, New York, NY, USA, 121–129. DOI: https://doi.org/10.1145/2338714.2338736

Google. 2017. Introducing Google Draco. (January 2017). https://opensource.googleblog.com/2017/01/introducing-draco-compression-for-3d.html.

Chris Greenhalgh and Steve Benford. 1995. MASSIVE: a distributed virtual reality system incorporating spatial trading. In *Distributed Computing Systems, 1995., Proceedings of the 15th International Conference on*. IEEE, 27–34.

Khronos Group. 2015. gltf 1.0 - Runtime 3D Asset Delivery. (October 2015). https://www.khronos.org/gltf.

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 Conference on 3D Web Technology (Web3D '14)*. ACM, New York, NY, USA, 35–43. DOI: https://doi.org/10.1145/2628588.2628589

Zhihan Lu, Paul Guerrero, Niloy J. Mitra, and Anthony Steed. 2016. Open3D: Crowdsourced Distributed Curation of City Models. In *Proceedings of the 21st International Conference on Web3D Technology (Web3D '16)*. ACM, New York, NY, USA, 87–94. DOI: https://doi.org/10.1145/2945292.2945302

Christophe Mouton, Samuel Parfouru, Clotilde Jeulin, Cecile Dutertre, Jean-Louis Goblet, Thomas Paviot, Samir Lamouri, Max Limper, Christian Stein, Johannes Behr, and Yvonne Jung. 2014. Enhancing the Plant Layout Design Process Using X3DOM and a Scalable Web3D Service Architecture. In *Proceedings of the 19th International ACM Conference on 3D Web Technologies (Web3D '14)*. ACM, New York, NY, USA, 125–132. DOI: https://doi.org/10.1145/2628588.2628592

Manuel Olbrich. 2016. Connecting VT RDF Resources to X3DOM. In *Proceedings of the 21st International Conference on Web3D Technology (Web3D '16)*. ACM, New York, NY, USA, 37–41. DOI: https://doi.org/10.1145/2945292.2945314

T. Parisi and R. Arnaud. 2011. 3D REST 3D specification v0.2. (April 2011). http://rest3d.org.

Michael Rymaszewski. 2007. *Second life: The official guide*. John Wiley & Sons. ISBN-10: 047009608X.

Andreas Schiefer, René Berndt, Torsten Ullrich, Volker Settgast, and Dieter W. Fellner. 2010. Service-oriented Scene Graph Manipulation. In *Proceedings of the 15th International Conference on Web 3D Technology (Web3D '10)*. ACM, NY, USA, 55–62. DOI: https://doi.org/10.1145/1836049.1836057

Arne Schilling, Jannes Bolling, and Claus Nagel. 2016. Using glTF for Streaming CityGML 3D City Models. In *Proceedings of the 21st International Conference on Web3D Technology (Web3D '16)*. ACM, New York, NY, USA, 109–116. DOI: https://doi.org/10.1145/2945292.2945312

Timothy Scully, Jozef Doboš, Timo Sturm, and Yvonne Jung. 2015. 3Drepo.Io: Building the Next Generation Web3D Repository with AngularJS and X3DOM. In *Proceedings of the 20th International Conference on 3D Web Technology (Web3D '15)*. ACM, New York, NY, USA, 235–243. DOI: https://doi.org/10.1145/2775292.2775312

Timothy Scully, Sebastian Friston, Carmen Fan, Jozef Doboš, and Anthony Steed. 2016. glTF Streaming from 3D Repo to X3DOM. In *Proceedings of the 21st International Conference on Web3D Technology (Web3D '16)*. ACM, New York, NY, USA, 7–15. DOI: https://doi.org/10.1145/2945292.2945297

Kristian Sons, Felix Klein, Dmitri Rubinstein, Sergiy Byelozyorov, and Philipp Slusallek. 2010. XML3D: Interactive 3D Graphics for the Web. In *Proceedings of the 15th International Conference on Web 3D Technology (Web3D '10)*. ACM, NY, USA, 175–184. DOI: https://doi.org/10.1145/1836049.1836076

Christian Stein, Max Limper, and Arjan Kuijper. 2014. Spatial Data Structures for Accelerated 3D Visibility Computation to Enable Large Model Visualization on the Web. In *Proceedings of the 19th International ACM Conference on 3D Web Technologies (Web3D '14)*. ACM, New York, NY, USA, 53–61. DOI: https://doi.org/10.1145/2628588.2628600

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 (Web3D '16)*. ACM, NY, USA, 117–125. DOI: https://doi.org/10.1145/2945292.2945293

Jan Sutter, Kristian Sons, and Philipp Slusallek. 2014. Blast: A Binary Large Structured Transmission Format for the Web. In *Proceedings of the 19th International ACM Conference on 3D Web Technologies (Web3D '14)*. ACM, New York, NY, USA, 45–52. DOI: https://doi.org/10.1145/2628588.2628599

Unity Technologies. 2016a. Coroutines. https://docs.unity3d.com/Manual/Coroutines.html. (2016).

Unity Technologies. 2016b. Getting started with WebGL development. (November 2016). https://docs.unity3d.com/Manual/webgl-gettingstarted.html.

Unity Technologies. 2016c. Loading Resources at Runtime. https://docs.unity3d.com/Manual/LoadingResourcesatRuntime.html. (2016).

Unity Technologies. 2016d. WebGL, CORS and XMLHttpRequest.withCredentials. (October 2016). https://forum.unity3d.com/threads/webgl-cors-and-xmlhttprequest-withcredentials.438004/.

Web3D Consortium. 1997. Information technology – Computer graphics and image processing – The Virtual Reality Modeling Language (VRML) – Part 1: Functional specification and UTF-8 encoding. (1997). International Standard ISO/IEC 14772-1:1997.

Web3D Consortium. 2013. Information technology – Computer graphics, image processing and environmental data representation – Extensible 3D (X3D) – Part 1: Architecture and base components. (2013). Internation Standard ISO/IEC 19775-1:2013.