Towards Declarative 3D in Web Architecture

Jean Le Feuvre
Telecom ParisTech; Institut Telecom; CNRS LTCI
46, rue Barrault 75634 PARIS CEDEX 13
jean.lefeuvre@telecom-paristech.fr

ABSTRACT
The recent WebGL integration in major web browser has open the way to many 3D applications as well as high-level libraries targeting 3D content developers. While most of these libraries provide solid grounds for interoperable 3D on web browsers, one might wonder if their use could not be simplified both in terms of processing overhead and 3D description syntax; looking beyond these issues, if there is room for a declarative 3D language for web architecture, its features should be well defined to ensure its success. In this paper, we review some use cases, some existing technologies and some drawback of existing tools in order to derive some requirement for the upcoming declarative 3D language for the HTML ecosystem.

Categories and Subject Descriptors
H.5.2 [INFORMATION INTERFACES AND PRESENTATION]: User Interfaces – *Graphical user interfaces (GUI), Standardization, Windowing Systems.*

General Terms
Standardization, Languages.

Keywords
Declarative, 3D, mixed 2D and 3D, WebGL, Stereoscopic Displays.

1. INTRODUCTION
Over the last twenty years, a growing number of technologies for describing, animating and controlling 3D objects or 3D worlds have appeared, and sometimes disappeared. Whether imperative or declarative, most of these technologies have had success in some market areas, but it is hard to identify the "big winner": the one technology to be used in any business environments. With the growing importance of the Web architecture as an underlying platform for many applications and market places, enabling 3D on the web has become a major part of recent web developments. The most noticeable 3D "newcomer" in the web is with no doubt WebGL [1], enabling web browsers a fast yet simple access to the device's GPU through the OpenGL ES 2.0 API [2]. Many interesting projects have been launched around this powerful API, using imperative approaches through JavaScript (JS), like the promising *GLGE*, *SceneJS*, *Three.js* or *PhiloGL*. Declarative approaches have also surfaced; we can cite X3DOM, an X3D implementation in JavaScript, or XML3D, a JS implementation of a 3D scene graph closely related to web concepts of HTML and CSS. It is worth notifying that even imperative approaches, such as game engines, usually require some declarative way of expressing the 3D models or levels design, and declarative approaches can already be seen in most systems, using XML or JSON parsing with XMLHttpRequest [3]. This paper does not aim at describing the different solutions already available [4] for integrated Web and 3D, nor to start yet another discussion on declarative versus imperative approaches: each solution has its pros and cons, but each might be needed depending on the application requirements. This paper will therefore attempt to focus on requirements that would make a browser-native declarative 3D support more appropriated than existing JS-based solutions.

As part of its research work on scene description technologies, Telecom ParisTech multimedia lab has developed GPAC [5], an open-source multimedia player. The research topics cover mainly 2D scene descriptions such as SVG or BIFS; it also covers some 3D aspects, through VRML based technologies such as X3D or BIFS. One specific topic of this work was on integrating these different scene representation technologies within a single graphics engine and mixing them in one multimedia presentation. This work was demonstrated in [6]. The purpose of this paper is to share some of the experience acquired during the development of this hybrid 2D/3D renderer, along with some more requirements derived from academic work related to this topic. These requirements are intended to be generic and uncorrelated with final syntax and future design choices such as handling of animations or usage of CSS.

This paper is organized as follows: in Section 2, we will briefly advocate for declarative 3D versus existing tools, and draft a first set of basic requirements. In Section 3, we will investigate some specific requirements around the topic of mixed 2D and 3D; in Section 4, we briefly investigate some aspects of multi-view rendering for auto-stereoscopic displays and derive some requirements for the upcoming Declarative 3D task. Section 5 finally concludes this paper.

2. Advocating for Declarative 3D
2.1 On scene graph
Virtual worlds are usually complex 3D environments with a large number of independent objects presented together on the screen. Whether each object is made of a single data structure (or node) or of a collection of structures, all 3D engines manipulate the collection of objects as a graph representing the scene to display, or *scene graph* in the usual terminology. This graph describes the relationship between objects, with more or less details. The basic level will be description of spatial relationships (transformation matrices), but complex scene graphs may also include interactivity relationship (scripting), temporal relationships (animations), physics relationships (collision, material elasticity...). Obviously, the more information a scene graph provides on the objects in the scene, the more complex and time consuming the rendering of the scene may grow. Scene graph is an important part of the interactive application logics, as it is usually the place where all software optimizations are done, such

WWW2012, April 16–20, 2012, Lyon, France.

as matrix stack handling, object picking, partial traversal of the graph... Benchmarks done in [7] show that existing JS 3D engines have hard times competing with a native scene graph and OpenGL implementation, but we should however keep in mind that the JS libraries tested are generic purposes libraries, rather than on-purpose designed ones. This is maybe one of the most challenging areas for DEC3D: obviously, declarative 3D implies usage of a scene graph, however it shall have clear advantages over JS ones, whether JSON or XML or binary, in order to be attractive to application designer. Indeed, a native scene graph is "frozen", and the implementation is not in the hands of the developer: if some design of this scene graph does not suit his needs, he will likely move to a script-base approach. One way to avoid such situation is to ensure modularity of the scene graph design. The focus of this paper is not to dig into the specific features supported by the scene graph, as existing standards such as X3D already cover a broad set of common features for DEC3D. It should however be noted that DEC3D is intended for integration with Web technologies, and as such could use a CSS-oriented design for styling, transformations and script-less animations such as SVG animations. Such features should typically be made configurable in the scene tree to optimize rendering routines, discarding for example the CSS inheritance phase or the animation module. From this remark, the following requirements are derived:

REQ1: DEC3D scene graph shall be modular; in particular, it shall allow an author to turn off unneeded features from the graph itself during the traversing of the scene tree (e.g. lighting, color transformations, collision detections, animations...), while still allowing for dynamic modifications of desired features,

REQ2: DEC3D scene graph shall be extensible, in particular it shall allow an author to design its own nodes; this should be done either programmatically or through Proto/XBL concepts.

2.2 On 3D Models

While WebGL provides direct and fast access to the GPU, most existing WebGL frameworks need to handle the objects they are rendering by themselves. This includes object geometry (polygons, triangles sets/fans/...), appearance (material and texture), positions (camera and model transformations), lighting and other graphical effects (shadows, particle systems...). Once these properties are assigned to an object, rendering is achieved through WebGL in near native speed. Most if not all these properties are loaded and manipulated in JavaScript, which can cost time. The loading of this properties from a model description (OBJ, Collada...) relies, when done in JS, on *XHR* [3] for text-based description (JSON, XML...), and additionally *ByteArray* objects for binary-coded models such as MPEG-4 3DMC ones [8]. Reaching high performances with such JS APIs remains challenging, as shown in [9], JS increasing the load time of very complex models as used in CAO or medical applications. Having a native support for model importing will drastically reduce loading times of many models; such a feature should however retain compatibility with pure WebGL imperative programming, in order to respect the specific needs of the application developer. We can therefore derive the following requirements:

REQ3: DEC3D shall support native loading of various model types, either textual or binaries, from any local or remote location; an appropriated MIME type should identify model formats,

REQ4: DEC3D shall define ways for natively loaded models to be used in a WebGL environment; for example, API/ID to retrieve the WebGLBuffer from the model and reference it in a shader program.

2.3 On WebGL and DEC3D

As stated previously, it is likely that relying only on a declarative scene graph may not suits the designer needs, for example when some default rendering algorithm in the DEC3D language cannot be easily expressed in a declarative way (dynamic shader design...). In the same way that OpenGL ES moved from a hard-wired graphical pipeline interface to a programmable-only GPU control, we believe that DEC3D should take into account the possibilities of unthought-of use cases and provide WebGL fallback to the developer; this will ensure a future-proof, flexible design and will encourage authors to use the language. This can be expressed by the following requirements:

REQ5: DEC3D shall allow an author to use only some native functionalities of the scene graph, for example object picking, while overriding other functionalities with WebGL code, for example drawing;

REQ6: DEC3D shall allow an author to use some native functionalities of the scene graph in parts of the scene tree while using custom behavior in other parts through WebGL callbacks.

3. Integration of 2D and 3D

One thrilling aspect of DEC3D is its usage in scenarios where 2D (HTML, SVG) and 3D (DEC3D, WebGL) objects are used at the same time, and communicating with each other. When designing an integrated renderer for SVG/BIFS/X3D, we have faced some issues that DEC3D could be confronted to, which are detailed in this section.

3.1 Rendering Contexts for 2D and 3D

Integrating 2D and 3D descriptions in an HTML scene can seem straightforward at first glance, but raises the same design issues as integration of SVG in HTML: HTML is a flow-layout scene description based on relative positioning of blocs or boxes, and is not well suited to host absolute positioning languages in its flow. The usual approach to solve this problem is to define a rendering region, similar to canvas, where the hosted language paints itself. This is for example the case when integrating SVG in HTML, one cannot simply insert an svg <*circle*> element in the flow, it has to be inside an <*svg*> element assigning a local coordinate system and bounds for the drawing area, in order to perform the HTML flow layout. Note that the bounds do not necessarily have to define a clipping area, e.g. the hosted content could be drawn outside this area. This approach is very similar to the canvas approach, where the size of the canvas region is exactly defined in terms of CSS dimensions so that flow layout can happen.

REQ7: DEC3D shall support drawing of 3D shapes and scene elements within the HTML flow layout, and shall not enforce the entire scene management to be in a 3D context.

On the other hand, some applications may wish to be full-window or full-screen 3D application, with no HTML layout above the 3D part. This is typical in games and virtual worlds, but other use cases may require this.

REQ8: DEC3D shall support using the entire HTML window as its 3D rendering area

3.2 Events and Coordinate System

The major inconvenience when handling a document mixing 2D and 3D content is the event system. The event system defines how

user events (mouse, keyboard, HMI devices), network events or other notification events are handled in a scene graph. Unfortunately, each standard has its own way of defining its own event system, and most of the time these are not compatible. VRML/X3D uses types events following the node field data types, and *ROUTE* mechanism to copy events from their source to any destination desired; events are generated by dedicated UI sensor nodes, such as *TouchSensor* or *ProximitySensor*. SVG and HTML use the DOM Event model, where events are generated with no explicit sensor but rather "appear" at any visible/geometry node and bubble up the scene graph from this node to the root node. These events are not typed in terms of XML data types, but have an IDL definition allowing manipulation of these events in script. Without scripting, interactivity is much more limited. In order to allow a simple design of the application mixing 2D and 3D content, we can add the following requirement:

REQ9: DEC3D shall use the DOM event model in order to cohabit with SVG or HTML applications.

Note that this requirement does not exclude usage of existing VRML/X3D sensors such as *ProximitySensor* or *SphereSensor*, but will rather transform them into grouping nodes catching simple mouse or keyboard events and firing new, 3D-specific events if desired.

Another issue faced with 2D/3D integration is the handling of coordinate systems. By default, most 2D languages use a raster-aligned coordinate system, with the origin (0,0) at the top-left of the canvas and the Y-axis going downwards; on the opposite, most 3D languages use a 3D Cartesian coordinate system, with the origin (0,0) at the center of the canvas, the Y-axis going upwards. While the handling of such differences is annoying for the implementation (Y scaling and translations happening all over the place), it becomes even trickier for the application designer. DOM-based 2D scene representations do not expose hit coordinate at hit point. On the opposite, 3D scene representation events usually carry much more information than screen and client coordinate. Getting hit point coordinate in 3D space is a basic use case, and getting the value of the normal or the texture coordinate at the hit point is also common when dealing with interactive textures. Scripting approaches such as *getScreenCTM* in SVG are clearly not sufficient to compute these details, as they would require computing in JS mouse ray and shape intersection to compute this data, and insert flip/translation matrix when switching between DEC3D and SVG. In order to simplify handling of clicking on / picking of shapes in an application mixing 2D and 3D, a unified system for retrieving hit coordinates in the local coordinate system:

REQ10: DEC3D shall use a coordinate system for events aligned with DOM Event coordinate system and provide a simple way of accessing pointing device coordinates in the local coordinate system.

REQ11: DEC3D shall have support for hit point coordinate, texture coordinate and normal value at hit in a DOM Event compatible way; this feature should only be enabled when advanced interaction is required.

3.3 Offscreen Rendering

One of the most compelling use case for 2D and 3D integration is getting more and more widespread in window manager of various operating systems through the terminology "Compositing" or "Composite Desktop": being able to use the output of any application as a texture for another application. WebGL allows for this by using the HTML Canvas object in 2D mode for texture creation, then passing the texture to GPU through WebGL's glTexImage2D. Note however that drawing web content into a 2D canvas is not allowed in most browsers, hence not yet interoperable.

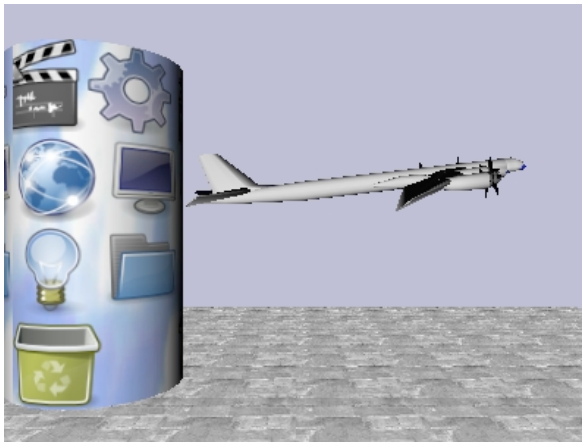

Figure 1 - Integration of SVG menu, X3D model and MPEG-4

There are endless possibilities with the ability to transform part of a sub-tree into a texture usable in 2D or 3D contexts, especially for non-linear transformations. Having a declarative mean to define such textures / offscreen rendering areas feel quite intuitive, as using WebGL and JS to implement such simple data transfers to GPU texture units seems quite an overhead. It should be noted that such features are present in MPEG-4 BIFS, through *CompositeTexture* nodes, as shown in Figure 1. These elements also allow for interactions and react to mouse and keyboards events. Existing layering elements such as HTML * * or inner *<svg>* could be a base for such a design.

REQ12: DEC3D shall have support for simple definition of offscreen rendering areas for 2D or 3D DOM content, and reuse of these areas as 3D textures or 2D patterns in SVG.

REQ13: DEC3D shall have support for offscreen rendering of part of the DOM tree, with support for DOM events in these sub trees.

4. 3D Displays

The past few years have seen the regain of interest for 3D entertainment using the human binocular vision system. 3D displays are becoming more and more widespread, whether for mobile devices (phones, portable gaming devices) or for home entertainment (TV, picture frames...).

The current focus of the industry is to achieve interoperable playback of video on these devices, through a various set of standards ranging from frame packing in AVC (two views in side-by-side or top-and-bottom packed in one frame), to more modern approaches combining video and depth / disparity maps, in order to generate multiple views from arbitrary viewer positions, as shown in Figure 2.

The next logical step will be to achieve interoperability of applications using such displays, for any possible characteristics of the display such as the number of views available or the optimal viewing distance. We therefore think the following requirement is reasonable:

REQ14: DEC3D shall have support for 3D displays and auto-stereoscopic interactive services.

Figure 2 - Five-views synthesis from video and depth

4.1 Depth for 2D

Existing 2D scene descriptions such as HTML or SVG usually work with fixed z-order in the scene tree, which can be altered through scripting mechanism by removing objects and inserting them back at the desired layer. These languages typically follow the painter algorithm when drawing their shapes, and do not take into account any depth information: the nodes are drawn in the order they are found in the scene tree. While this model is fine in 2D space, it is no longer appropriated when designing interfaces for 3D displays, where depth (or z) is an inherent dimension of the service, as are horizontal and vertical positions. On the other hand, defining a complete 3D rendering context for the sole purpose of displaying an HTML button with a depth effect (screen pop-out, back and forth bouncing at the screen surface) seems quite an overhead for the author. This situation will only get worse if a 2D area with a depth effect also has a 2D sub-area with another depth effect. Simple extensions such as depth / z offset and scaling for 2D objects will be sufficient for most effects, but more powerful tools such as CSS 3D transforms could also help here.

REQ15: DEC3D shall support simple ways of assigning a depth or z value to a 2D HTML or SVG area; depth values shall be cumulated in a hierarchical way, as are regular 3D transformation matrices.

Another interesting feature in the years to come will be the ability of the device hardware to use depth-image along with texture data to generate image-dependent viewpoints. Depth-data handling also make its ways into UI systems with devices such as the Microsoft Kinect, and it won't be long until TV are equipped with such cameras. This naturally leads to believe that introducing DIBR (Depth-Image Base Rendering) into DEC3D seems an interesting path.

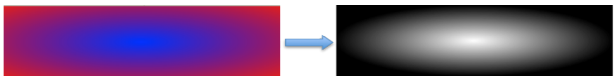

Figure 3 - Synthesizing depth map from SVG gradients

As explained in [10], we believe that using SVG or canvas 2D to generate texture data that could be used as depth data for other 2D objects, through simple component transfer rules as shown in Figure 3, is a powerful way of authoring transition effects for 3D displays.

REQ16: DEC3D shall have support for Depth-Image Based Representation, in order to allow for multiple view generation of 2D objects or areas in the content.

REQ17: DEC3D shall be able to generate synthetic depth maps from the different graphical primitives in the content, whether 2D or 3D, and whether defined in DEC3D or external to its namespace.

4.2 Virtual Camera Calibration

One important aspect of rendering for 3D displays is that the depth effect may not be exactly tied to the perspective settings of the 3D environment, and authors may decide to center a dedicated object on the screen plane, or before or behind the screen, without changing virtual camera settings. In other words, an author may decide to change the vergence point of the different cameras used during multiview rendering passes. Moreover, most of existing 3D languages do not take into account camera parameters for multiview rendering, such as camera displacement between views (circular, linear, off-axis), which may be modified by an author depending on its application type. This leads to non-interoperability between implementations. If DEC3D includes support for multi-view displays, it must therefore fulfill the following requirement:

REQ18: DEC3D shall be able to define the camera parameters used during multi-view generation, such as for vergence point (screen plane) location or camera displacement type.

5. Conclusion

In this paper, we have exposed our views on some aspects a declarative 3D language for web architecture should cover. More specifically, we have reviewed some of the difficulties encountered during the development of a mixed 2D and 3D multimedia renderer. We have also exposed some limitations in existing declarative technologies when designing content for auto-stereoscopic displays. Based on this analysis, we have derived some requirements for such a language and hope to contribute, in the near future, to DEC3D activity, both in terms of requirements and developments.

6. ACKNOWLEDGMENTS

Part of this work has been financed by the French-funded ANR project CALDER.

7. REFERENCES

[1] WebGL, http://www.khronos.org/webgl/

[2] OpenGL ES 2.0, http://www.khronos.org/registry/gles/

[3] XHR, http://www.w3.org/TR/XMLHttpRequest/

[4] http://www.khronos.org/webgl/wiki/User_Contributions

[5] Le Feuvre, J., Concolato, C., and Moissinac, J. 2007. GPAC: open source multimedia framework. In Proceedings of the 15th international Conference on Multimedia (Augsburg, Germany, September 25 - 29, 2007). MULTIMEDIA '07.

[6] Concolato, C. and Le Feuvre, J. 2008. Playback of mixed multimedia document. In *Proceeding of the Eighth ACM Symposium on Document Engineering* (Sao Paulo, Brazil, September 16 - 19, 2008). DocEng '08. ACM, New York, NY, 219-220. DOI= http://doi.acm.org/10.1145/1410140.1410185

[7] http://granular.cs.umu.se/browserphysics/?p=7

[8] B Jovanova, M Preda, and F Preteux, “Mpeg-4 part 25: A graphics compression framework for xml-based scene graph formats,” Signal Processing: Image Communication, vol. 24, pp. 101–114, 2009.

[9] http://blog.n01se.net/?p=248

[10] http://svgopen.org/2010/papers/54-SVG_Extensions_for_3D_displays/