Declarative AR in the Web with XML3D and Xflow

Felix Klein
Intel Visual Computing Institute
klein@intel-vci.uni-saarland.de

Philipp Slusallek
Intel Visual Computing Institute
DFKI
slusallek@dfki.de

Abstract

Augmented Reality is about enhancing video streams from the real world with dynamic, interactive, rich-media content. We propose that HTML5 browsers (the de-facto runtime-engines and viewers for dynamic, interactive, rich-media 2D environments) are becoming ideal AR engines – when extended with support for dynamic and interactive 3D content as well as efficient image and computer vision processing. Early but fully functional prototypes have already been developed using our XML3D and Xflow technology and will be demonstrated.

Modern Web technologies already provide native support for hardware-accelerated 3D graphics (WebGL) as well as access to cameras and many other sensors. Recently, JavaScript implementations in browsers became significantly faster (e.g. with asm.js) and upcoming extensions like WebCL (Khronos) or Parallel-Javascript/RiverTrail (Intel/Mozilla) promise to make massively parallel computing readily available within the browser as well. This enables efficient image processing, feature detection, and tracking, but also real-time animations and other compute intensive processing – right within the browser.

As a result, it is now becoming possible to design Web-based AR applications that completely run in any modern Web browser without extensions or plug-ins. Such AR applications can take full advantage of the browsers rich-media environment and benefit from the Web’s huge ecosystem, such as cloud computing, service-oriented architectures, Open Linked Data, and many more. This would enable millions of Web Developers to easily integrate AR technology directly into their Web applications. Web browsers will provide AR at your fingertips: A single click on a URL can fully immerse the user in arbitrary AR experiences even on mobile devices.

We show how XML3D, an extension to HTML5 for declarative 3D graphics, and Xflow, a declarative language for dataflow processing, provide a high-level framework to develop AR applications on the Web, while encapsulating and abstracting from low-level image and AR processing. Our approach does not provide functionality as a large, monolithic library, but rather as a collection of smaller building-blocks that can flexibly be combined and reused in many ways depending on the needs of applications and developers. We also show how AR operators can be accelerated in the browser with emerging Web technologies such as Parallel JavaScript (RiverTrail).

Keywords: Multiport Decomposition, Parallel processor, simulation.

1 Introduction

The Web was originally designed as an information medium, with the simple task to display formatted text and media, such as images. With the rise of modern Web technologies the Web further evolved into a ubiquitous application platform. Not only does the Web now natively support hardware-accelerated 3D graphics with WebGL, it also provides access to cameras via the WebRTC API and several options to perform image processing and visual analysis. All these technologies combined, the Web turns out to be a great platform to provide AR at your fingertips. A simple link can send a user to a Web page with embedded AR functionality, running without any browser extension or plug-in. Such AR applications work in any
modern Web browser, greatly simplifying the process of deploying applications on a very heterogeneous set of devices, increasingly also including tablets and mobile phones.

Another advantage of the Web in general is the accessibility of the technology. Once AR technology is easily available in browsers, millions of Web developer could pick up the technology and start integrating AR into their applications. However, there are still a number of limitations before this becomes a reality. Several of new technologies required for AR, such as WebGL, focus on bare functionality, neglecting interoperability with other, well-known Web technologies such as HTML and CSS. As a result, there is a wide gap between traditional, declarative Web design based on HTML/CSS and 3D Web application development. The later is based on imperative low-level APIs such as WebGL or specialized JavaScript libraries such as three.js [3] that are mostly isolated from other Web technologies. Other emerging technologies, such as Parallel JavaScript or asm.js, only improve the capabilities of iterative scripts running inside an otherwise mostly declarative Web. Consequently, creating even a basic augmented reality (AR) application on the Web requires Web developers to learn a stack of new languages or APIs entirely different from what they are familiar with.

It should not be forgotten, that the actual value of the Web is not in the JavaScript APIs like WebGL, WebRTC or others, that could or are equally be available outside of browsers. The key element of the Web is a common, structured, hierarchical, declarative, extensible data representation, the DOM, and its external representation as HTML5. It is this common structure that allows documents to be indexed and searched, allows powerful tools such as jQuery to be used on almost any data, and thus makes Web development so much more productive.

The Declerative 3D on the Web initiative at the W3C tries to make 3D Web development more accessible to Web developers by adding declarative 3D content to HTML5, making it a part of the structured Web. In this position paper, we discuss how an integration of AR inside of Declarative 3D provides a very intuitive framework for Web developers to create AR applications on the Web. We base our work on XML3D, a proposal for Declarative 3D, and Xflow, a declarative language for general purpose dataflow processing. We show how a minimal extension to these technologies results in a flexible AR framework that integrates nicely with a 3D scene graph. In addition, we discuss how several existing and emerging Web technologies can be used to improve the performance of AR on the Web.

2 XML3D

XML3D [4] has been designed as a minimal extension to existing Web technologies for declarative 3D content. It provides a light-weight scene graph in HTML5 with meshes, shaders, light sources, and viewpoints composed inside of a transformation hierarchy.

One important design consideration of XML3D is to keep core data structures of the scene graph generic. Instead of introducing specialized elements with custom data structures, XML3D provides a flexible approach to declare generic data blocks with typed values using the <data> element. When connected to the scene graph as child elements or through references, those data blocks are interpreted as input for meshes or shaders. This approach tries to match the generic design of today’s graphics APIs, which support programmable shaders and work on generic input buffers. Consequently, we have a direct mapping from HTML data to internal GPU buffers, eliminating most conversion overhead. A small example scene in XML3D can be seen in Figure 1.

The most recent version of XML3D is realized as a polyfill implementation based on JavaScript, internally using WebGL for rendering. [5]

3 Xflow

Modern 3D graphics relies on the programmability of graphics hardware for computationally expensive tasks, e.g. character animations, particle effects, or post processing. One approach to provide this functionality through Declarative 3D (one taken by X3D) is the implementation of specialized nodes for each use-case
Figure 1: A very basic 3D scene using most elements defined by the XML3D specification. `<shader>` and `<lightshader>` define surface properties (reflectance and emission) respectively, generically configured via child value elements (e.g. `float`, `float3`). `<light>` and `<mesh>` define light sources and geometry of the scene, while `<view>` specifies potential viewpoints of the camera. All of these elements can be placed inside a transformation hierarchy of `<group>` elements. Larger resources like geometry can be loaded from external files with different formats - such as with images and videos in HTML5.

(e.g. specialized nodes for character animations). However, the result of this approach is a blown-up specification, which requires high efforts to be fully supported by clients and still struggles to keep up with the flexibility of programmable graphics hardware. Thus, the actual challenge for Declarative 3D is to provide this functionality in a flexible way while at the same time being easier to use than the current low-level APIs and to avoid the impedance mismatch between the high- and low-level code. XML3D aims to achieve this with Xflow [2], a declarative language for dataflow processing.

Dataflows in Xflow are a combination of small, modular operators that process generic input data in the form of array buffers. Operators are designed to be reusable and can perform operations as basic as `addition` and `multiplication` and as complex as skeletal animations and more, allowing a great deal of flexibility when authoring the dataflow. However, the dataflow abstracts over many low-level aspects of data processing including memory management, efficient scheduling, and parallelization of the execution. Due to the functional description of the dataflow, Xflow can easily optimize and parallelize the execution using all available computational resources (e.g. multiple CPU and GPU cores). Similar to XML3D, Xflow has been realized as a polyfill implementation completely in JavaScript without plug-ins or other additions.

Figure 2 shows the declaration of a simple dataflow. Dataflows are modelled with a connection of `<data>` element. Input data is defined with generic value elements such as `<float>` and `<float3>`. Data is processed by attaching `operators` on a `<data>` node via the `operator` attribute.

So far, Xflow has been presented mostly for mesh processing [2] (e.g. for character animations). However, it is designed as a general-purpose processing tool. Consequently, Xflow is a good starting point to integrate advanced AR features into Declarative 3D, as we will describe in the following section.
Figure 2: An example for a dataflow in Xflow processing generic float data. In $data#morphedPos$ we compute a morphed position with the $xflow.morph$ operator using the $pos$, $pAdd$ and $weight$ values and returning a new $pos$ value overriding the original one. In $data#morphedPosAndNorm$ we also computed a morphed and normalized normal value using two operators. Note that we refer the result of $data#morphedPos$ inside of $#morphedPosAndNorm$ to be able to access the $weight$ value. The final output of $#morphedPosAndNorm$ includes the morphed normal and position, since all values are propagated upwards the hierarchy unless otherwise specified.

4 Declarative AR

In this section we outline how AR is realized with XML3D and Xflow in two ways: First we show a purely declarative approach connecting AR specific Xflow operators directly with the transformation hierarchy and viewpoint. Second we demonstrate a more flexible approach that connects scripts to the Xflow graph allowing for arbitrary DOM modifications

4.1 Document Connection

An AR operator, which places 3D objects relative to detected markers of a video stream, needs to influence three aspects in the scene graph:

1. The transformation hierarchy of objects connected to the marker,
2. The visibility of those objects and
3. The projection matrix of the current view point to match the intrinsic camera transformation of the video stream.

We start by implementing a basic AR operator (xflar.detect) based on JSARToolkit. This operators takes as input the video stream and a list of markers. It outputs a list of transformations and visibility flags (one for each marker) and a perspective transformation for the view point:

```xml
<!-- AR data for all markers: -->
<data id="arBase" compute="transforms, seenFlags, perspective = xflar.detect(arvideo, markers)"
    <int name="markers">22 64</int>
    <texture name="arvideo">
        <video autoplay="true" src="videostream.ogg"></video>
    </texture>
</data>
```

1https://github.com/kig/JSARToolKit
In order to modify the transformation hierarchy of the XML3D scene graph via Xflow output, we allow the connection of a `<data>` element to a `<group>` node's transform attribute. The only requirement for the `<data>` element is to provide an output with name `transformation` and type `float4x4`, storing the coordinates of the transformation matrix:

```xml
<!-- Attach transformation to group -->
<group name="#obj1AR" shader="#obj1Shader"

  <!-- Content: Geometry placed relative to first marker -->
</group>
```

Next, we need to influence the visibility of objects, which might need to be hidden if the corresponding marker is not detected. For this we use a custom shader, which takes a boolean parameter for visibility. Now we simply forward the visibility flag of the Xflow output to the input of the shader:

```xml
<!-- Shader for first object, storing visibility -->
<shader id="obj1Shader" script="urn:xml3d:shader:phongvs"

  <float3 name="diffuseColor">1.0 0.4 0.2</float3>
  <float name="ambientIntensity">0.2</float>
  <!-- Take visibility from AR -->
  <data filter="keep(visibility)" src="#obj1AR" />
</shader>
```

Finally, in order to efficiently update the projection matrix of the current viewpoint to match the field of view of the video stream, we use the `perspective` attribute of the `<view>` element, referring to a `<data>` element containing a `float4x4` value of name `perspective`. This perspective transformation is then simply used for the viewpoint overriding the original perspective matrix corresponding to the declared content:

```xml
<!-- Viewpoint with connection to AR data -->
&view id="View" perspective="#arBase" />
```

### 4.2 Script Integration

The declarative connection described previously provides us an easy option to create basic AR application. However, for more complex applications we often want to adapt the scene in a very dynamic fashion depending on appearing and disappearing markers (e.g. an application where an object is always moved to the most recently appeared marker). For these kind of applications, declarative descriptions get too complex and violate a design principle of XML3D to keep interactive components out of the declarative format. Instead, a more flexible approach is to integrate scripts in the augmented reality application, that can access the augmented reality result and modify the XML3D document correspondingly. As these scripts are usually simple and operate on small data sets (the transformations due to a visible marker), they should not slow down the application significantly.

We implement the `XML3DDataObserver` class in order to integrate scripts inside an XML3D and Xflow application. This observer is designed similar to the MutationObserver [6] of the DOM and allows applications to observe the complete set or subsets of the dataflow results (filtered via name sets) from arbitrary `<data>` elements. The callback of the observer is invoked whenever any of the observed results changes. However, just as with MutationObservers, the callback is not invoked immediately on each change of the Xflow graph, but only at most once per frame, providing a list of all changes. This design allows us to integrate
the callback effectively within the render cycle of XML3D as can be seen in Figure 3. This integration allows us to perfectly synchronize dataflow execution, callback invocation, and rendering such that modifications in the callback are immediately visible in the following frame.

4.3 Improved AR Integration

For a first prototype, we implemented a fairly monolithic AR operator, simply wrapping JSARToolKit. A proper AR integration into Xflow would consist of several small, generic operators that implement different steps of the marker detection pipeline. This design has the general advantage to be more flexible and modular. That way, new AR related algorithms can be easily implemented and plugged into the framework. To demonstrate the flexibility of combining generic operators, we implemented an additional set of operators analyzing the video stream to determine the optimal threshold for the marker detection algorithm of JSARToolKit:

```xml
<data compute="transforms,seenFlags,perspective = xflar.detect(video, markers, threshold)"
    <int name="markers">1 2</int>
</data>
<data compute="threshold = xflar.getOtsuThreshold(histogram)"
    <data compute="histogram = xflip.createHistogram(grfrm,channel)"
        <int name="channel">0</int>
    </data>
</data>
</data>
</data>
</data>

<video>
</video>
</texture>
</data>
</data>
</data>
</data>
</data>
```
<table>
<thead>
<tr>
<th>Xflow Operator</th>
<th>Time in ms</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequential</td>
<td>Parallel</td>
</tr>
<tr>
<td>Convolution 3x3</td>
<td>236</td>
<td>52</td>
</tr>
<tr>
<td>Convolution 5x5</td>
<td>502</td>
<td>89</td>
</tr>
<tr>
<td>Convolution 7x7</td>
<td>880</td>
<td>141</td>
</tr>
<tr>
<td>Convolution 9x9</td>
<td>1420</td>
<td>218</td>
</tr>
</tbody>
</table>

Table 1: Performance improvements of parallelizing image processing operators with Parallel JavaScript.

We applied a convolution with different kernel sizes on a video with a resolution of 896x512. The execution time was measured on a machine with an Intel Core i7-2670QM CPU. We can see a significant speed-up ranging from a factor of 4.5 to 6.5. For larger kernel sizes, the impact of converting data structures for parallel execution decreases due to longer processing times, which results in a higher speed up.

This example uses xflip, a collection of Xflow operators for image processing. We use xflip.grayscale and xflip.createHistogram to get a histogram from the video stream. The xflar.getOtsuThreshold operator takes the histogram and produces a threshold value matching the lighting conditions. We finally pass the threshold to xflar.detect, improving reliability of the marker detection.

Another important aspect of the AR integration is the composition of the video stream and rendered content. Our most basic prototype simply places the 3D content on top of a `<video>` element, which has the problem, that video and rendered 3D content are not perfectly synchronized. With the XML3DDataObserver we can perfectly synchronize video stream and rendered 3D content, by accessing the video image through Xflow and rendering it inside a `<canvas>` element below the 3D content. However, the most effective and elegant approach to composite video stream and 3D content is through a post-processing step after rendering. This approach also allows for 3D content to be correctly placed before and after content of a video with depth information (e.g. via a Kinect sensor). Since Xflow can be connected with any part of the rendering pipeline, including post processing, this kind of integration should work very elegantly without the need for new, specialized features.

4.4 Harnessing the Web

Because of the tight integration with Web services we can utilize many other features of the Web for other types of AR applications. We can access GPS coordinates via the GeoLocation API and use a Web service to access relevant POIs that can be forwarded to the AR dataflow instead of markers. Displaying 2D labels over the video stream is trivial with HTML, which does not only provide basic 2D layout, but the whole Rich-Media engine, e.g. to display images, videos or whole web pages in addition to text.

4.5 Flexible Data Formats for 3D and AR

XML3D allows for an easy integration of many data formats due to its simple and generic core. Any type of 3D format simply needs to be converted to generic buffers, which are conform to hardware. Data encoding can be performed offline on the server or online on the client via Xflow, which even supports encoding integrated into the rendering pipeline, e.g. via the ImageGeometry approach. Since the generic data concept and Xflow are not limited to geometry, we can use the same approach to provide AR specific data, such as feature definitions for POIs.
5 Optimizations

AR applications include computationally expensive tasks such as image processing and visual analysis, which can be effectively parallelized to increase performance. Since JavaScript is focused on single-threaded code execution with little support for data-parallelism, we need to look at alternative/ emerging Web technologies in order to further improve performance of AR. One of these technologies is Parallel JavaScript, a data-parallel programming API for JavaScript proposed by Intel Labs [1]. Using Intel’s River Trail prototype, kernels written in a slightly restricted form of JavaScript can be translated into OpenCL for efficient execution on multi-core CPUs and GPUs. We optimized a number of image processing operators with Parallel JavaScript which resulted in some significant performance improvements as can be seen in Table 1.

Other upcoming Web technologies that can be considered to further improve performance of AR are WebCL, NaCL, and asm.js.

Finally, the execution of Xflow dataflows can be integrated into the rendering pipeline via vertex shader code, which provides another option to optimize AR, especially when connected with 3D graphics.

6 Results

We implemented two prototype AR applications based on JSARToolKit. The first application only makes use of the declarative connection to map a teapot on top of a visible marker. The second application makes use of the script integration to dynamically make a teapot jump from one visible marker to the other.

References


\[\text{http://xml3d.github.io/xml3d-examples/examples/xflowAR/ar_simple_no_flip.xhtml}\]

\[\text{http://xml3d.github.io/xml3d-examples/examples/xflowAR/ar_flying_teapot.xhtml}\]