Declarative AR and Image Processing on the Web with Xflow

Felix Klein∗1,2, Dmitri Rubinstein†1,3, Kristian Sons‡1,2,3, Farshad Einabadi§1, Stephan Herhut¶4, and Philipp Slusallek∥1,2,3

1Saarland University  2Intel VCI  3DFKI  4Intel Labs

Abstract

Recently, modern Web browser became capable of supporting powerful, interactive 3D graphics both via the low-level, imperative API of WebGL as well as via a high-level, declarative approach like XML3D. The obvious next step (particularly with respect to mobile platforms) is to combine video from the real world with matched virtual content – Augmented or Mixed Reality (AR/MR). However, AR requires extensive image or video processing, feature detection and tracking, and applying the results to 3D rendering – all of which is hard to implement in a Web context.

In this paper we present a novel approach that encapsulates low-level image-processing and AR operations into re-usable high-level XML3D/Xflow components that are part of the HTML-5 DOM. A Web developer can then easily and flexibly arrange these components into (possibly complex) processing flow-graphs without having to worry about the internal computations and the structure of these modules. Our extended Xflow implementation automatically optimizes, schedules, and synchronizes the processing of the flow graph(s) in the context of real-time 3D rendering. Finally, we provide an integration model that greatly simplifies building AR applications for the browser.

We demonstrate this with several simple AR and image processing applications using a polyfill implementation working in all modern browsers and evaluate the performance. Finally, we show how the declarative framework can be optimized with respect to performance and usability using parallelization with Web Workers and RiverTrail.

Keywords:  XML3D, dataflow, visualization, Web3D, WebGL

1 Introduction

The Web originally was designed as an information medium, with the simple task to display formatted text and media, such as images. While this purpose stands firm, with the rise of cutting-edge 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. Using all these technologies, we can develop augmented reality applications with embedded 3D graphics running in any modern browser without the need to install a plug-in or extension.

However, this quick development of the Web came at a price. Several new technologies such as WebGL were designed with a focus on bare functionality, neglecting interoperability with other, traditional 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 that are mostly isolated from other Web technologies. 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.

The Declarative 3D on the Web initiative at the W3C tries to make
3D Web development more accessible to Web developers by providing a declarative language for 3D content, just as HTML does for 2D layout and SVG for vector graphics. However, a challenge of Declarative 3D is the flexible integration of high-performance computations that are inherently bound to interactive 3D graphics. These computations include mesh animations, particle effects, post-processing effects, and last but not least: image processing.

In this paper we focus on the combination of Declarative 3D with image processing and AR. 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 image-processing framework that integrates nicely with a 3D scene graph, but can also be used separately. We further demonstrate how this framework can be used to create simple AR applications with ease.

We start with related work in image processing and Declarative 3D in Section 2. In Section 3, we show how a minimal extension to Xflow is sufficient to support processing of images and videos. We continue with AR in Section 4, which requires a small extension to the previous XML3D and Xflow system. In Section 5 we inspect how we can use parallelization to optimize our image-processing implementation with respect to performance and usability. Finally, we show several applications built with our system in Section 6 with conclusions and future work in Section 7.

2 Related Work

As our goal is to combine Declarative 3D with image processing, we will discuss related work in both areas.

2.1 Image Processing

Image processing is any form of signal processing that takes one or several images as input, producing other images or extracting arbitrary parameters from the input. Computer vision uses image processing specifically to extract (high-level) characteristics from the image. For both fields exists an enormous amount of literature and several available toolkits. In the following, we first determine the requirements of image processing and computer vision in general. Afterwards we inspect today’s options to perform image processing on the Web.

2.1.1 OpenCV

OpenCV [OpenCV Foundation 2012] is one of the most comprehensive real-time image processing and computer vision libraries available today. It is open source and developed in C/C++. Therefore, it can be easily compiled and used in today’s most common desktop and mobile operating systems. Application areas of OpenCV are very wide, including, but not limited to: human computer interaction, AR, mobile robotics, face, object and gesture recognition etc.

However, C/C++ libraries such as OpenCV cannot be used directly in the browser. They would have to be included as plug-ins, requiring separate installation, which is not desirable.

2.1.2 Canvas Based Image Processing

Since the introduction of the <canvas> element, it is possible to process images with JavaScript on a per-pixel level. The canvas 2D API [W3C 2006] provides functions to draw basic shapes, polygons, text, and other images onto the canvas with a set of different blending operators. In addition, the canvas 2D API also provides direct access to the pixel data of the canvas via getImageData(). The pixel data is returned as a simple typed array that can be modified arbitrarily with JavaScript. While this option provides maximum flexibility, processing large images on a pixel by pixel basis with JavaScript tends to be slow and may not be applicable for real-time applications.

WebGL [Khronos Group 2011] brings hardware-accelerated 3D graphics to the canvas and also provides a way to read the result with readPixels(). With the possibility to use arbitrary shader code, WebGL is an efficient tool to perform certain types of image processing parallelized on the GPU. However, doing image processing with WebGL requires the correct setup of the rendering pipeline, uploading of textures, and the creation of low-level shader code.

With the canvas 2D API and WebGL, the Web provides two flexible options for imperative image processing, but both of them require a rather large amount of boilerplate code, especially if integrated with the Web document. Frameworks like processing.js [Salga et al. 2011] are built on top of canvas APIs to provide a very concise, imperative language to do basic, interactive image processing, effectively removing most boilerplate code. However, processing.js introduces an entirely new, imperative language to write image processing applications isolated from the rest of the Web page. This makes it not suitable for integration in more complex Web applications that require e.g. the communication between the processing.js components with Web page elements.

All of the canvas based image processing options ultimately suffer from being isolated from the DOM, which makes them incompatible with other Web technologies, such as events, and libraries, such as jQuery. However, they offer a great basis to build declarative image processing, while hiding the concrete implementation.

2.1.3 JSARToolKit

One of the most popular marker-based Augmented Reality (AR) libraries is ARToolKit [Kato et al. 1999]. Written in C and C++, it allows easy development of multi-platform AR applications using OpenGL for rendering. The library is fast enough for real time applications and is freely available as open source software. Because of its popularity the library has been ported to Java, C#, ActionScript3, Android (NyARToolKit), Flash (FLARToolKit), Silverlight (SLARToolkit) and JavaScript (JSARToolKit).

The JSARToolKit library allows to easily create AR applications for the Web. JSARToolKit in combination with the new getUserMedia API [W3C 2012d] allows writing AR applications with camera input in pure JavaScript [Heikkinen 2012]. However, in comparison with the original ARToolkit, JSARToolKit is limited in its functionality. Only ID markers are tested, camera calibration code is missing and produced matrices need to be converted in order to be used with WebGL.

2.1.4 Filter Effects

Filter Effects (FE) [W3C 2012c] is a proposal, which provides means to apply image processing to HTML and SVG elements. FE provides a predefined set of simple 2D image processing filters such as blurring, specular lighting, color compositing, blending, etc. A filter can be assigned to an element via CSS to process the element’s rendering. The result of a filter either replaces the original rendering or can serve as input for another filter. Thus it is possible to build filter graphs. Additionally, FE allows to use GLSL shaders on elements via the feCustom filter. For this filter, an implicit vertex grid is created based on the filter’s parameters. The vertex and fragment shader operations are applied on that grid, with the input images as shader parameters.
Filter Effects is a good example for a dataflow technology that is well integrated into Web technologies. It provides an easy way for Web developers to perform image processing, but – with the f- Custom shader – it is also possible for graphics experts to perform more complex operations. However, Filter Effects is designed for the very specific use-case of processing the appearance of DOM elements. For that reason, it only supports operators with a single output image and is thus not suitable for computer vision tasks, which require other output types. In addition, due to the limited depth model of HTML and SVG even filters that create depth will be projected back to the target layers after rendering and thus there is no real spatial interaction with surrounding elements.

2.2 Declarative 3D

Most content of the Web is still provided as HTML, SVG, and CSS and is thus declarative. It was with the introduction of the `<canvas>` element that the Web got an imperative option to produce visible content in a more flexible way. Shortly after that, WebGL was introduced to allow hardware-accelerated 3D graphics being rendered inside of the `<canvas>` element. Therefore, today’s Web provides declarative and imperative options for 2D graphics, but only an imperative option for 3D graphics.

Declerative 3D for the Web aims to extend the Web with a declarative option for 3D graphics that is well-connected with the Document Object Model (DOM), events, CSS, and many other Web technologies. One goal of this approach is to make 3D graphics more accessible to Web developers, who can reuse most of their knowledge when working with declarative 3D applications.

2.2.1 X3D and X3DOM

X3DOM [Behr et al. 2009] is a model, prototype, and framework to integrate X3D into HTML. It defines a new profile, which only contains those nodes that make sense in the Web context and thus streamlines X3D. But it keeps also many of the VRML/X3D concepts such as the DEF/USE, routing, inlining, etc. Those concepts are new to Web developers. On the other hand, it introduces many concepts and nodes that are not part of the X3D standard, for instance CSS transformations and new geometry formats such as `<imagegeometry>`.

X3DOM comes with an example on how to integrate FLAR-ToolKit. This connection is based entirely on DOM modifications via JavaScript and therefore not declarative. It is also limited by ignoring viewpoint calibrations depending on the used camera. This makes the example work only for cameras matching the predefined field of view.

Although X3DOM has a strong user community, we decided against X3DOM for the AR framework. Main reason for this is the approach of X3D to provide specialized nodes matching specific use-cases, often ignoring the possibility to reuse functionality. This results in the introduction of many specialized nodes and a blown-up specification, increasing the workload for implementations to achieve full support.

How the specialized approach of X3D applies to AR is particularly well seen from a review of current proposals for AR related nodes for X3D 1. Not only do the proposals discuss specialized nodes to access AR output, they also plan to introduce new nodes for video stream backgrounds and new viewpoints configurable with the AR output. Contrary to this approach, we aim to provide AR in a generic way that introduce as few new concepts and nodes as possible, lowering implementation overhead. In order to achieve this, we think it is essential to reuse existing Web technologies whenever applicable – very different than X3D, which still tries to provide all functionality independent of the Web.

2.2.2 XML3D

XML3D [Sons et al. 2010] is 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 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. Connected to the scene graph, those data blocks are used for instance for mesh or shader input. 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, reducing conversion overhead.

The most recent version of XML3D is realized as a polyfill implementation based on JavaScript internally using WebGL for rendering. [Sons et al. 2013]

2.2.3 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 major challenge for Declarative 3D is to provide this functionality in a flexible way that is easier to use than current low-level APIs and avoids the impedance mismatch between the high- and low-level code. If we simply cover this functionality with use-case oriented, specialized features, we will get a continuously growing specification struggling to keep up with evolving 3D graphics.

Instead of relying on fixed-functionality, XML3D takes a generic approach by using Xflow [Klein et al. 2012], 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.

Xflow [Klein et al. 2012] has been presented mostly for mesh processing (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 image processing into Declarative 3D, as we will describe in the following section.

3 Image Processing with Xflow

Image processing is connected to 3D graphics in several ways. Not only can image processing enhance the surface appearance of objects by modifying textures, it is also essential for computer vision combined with 3D graphics, e.g. augmented reality applications. In
To process new values we can attach operators. Maps are simply merged into the parent's map. Individual, named fields and other relevant for the requested result. Before an operator of a hierarchy of elements is evaluated, usually all child elements need to be evaluated first. However, note that Xflow does not simply use the element's subtree, skipping those not relevant for the requested result. Before an operator of a data element is evaluated, usually all child data elements need to be evaluated first. However, note that Xflow does not simply use the hierarchy of data elements to determine the execution order but analyses the exact connection of operator in- and output, which often allows for more parallelization among hierarchically ordered data nodes.

In order to properly explain the integration of image processing, we first explain the basic concepts of Xflow, i.e. how general dataflows are declared and executed.

Figure 2 shows the declaration of a simple dataflow. A dataflow in Xflow is a combination of data elements, which are either connected through the DOM hierarchy or via id references with the src attribute. Each data node defines a named value map declared through the node's children. Those children come in two types: typed ValueElements, such as float or float3, defining individual, named fields and other data elements, whose value maps are simply merged into the parent's map.

To process new values we can attach operators to data elements via the compute attribute. Operators are invoked just like regular JavaScript functions: we specify input parameters inside the parenthesis, referring data of the value map by name, and provide names for the output values to the left of the assigned operator. The output of the operator always overrides original values with the same name.

The declared dataflow is executed by requesting the named value map of a data element, which triggers the execution of all operators attached in the data element's subtree, skipping those not relevant for the requested result. Before an operator of a data element is evaluated, usually all child data elements need to be evaluated first. However, note that Xflow does not simply use the hierarchy of data elements to determine the execution order but analyses the exact connection of operator in- and output, which often allows for more parallelization among hierarchically ordered data nodes.

3.1 Introduction to Xflow

In order to properly explain the integration of image processing, we first explain the basic concepts of Xflow, i.e. how general dataflows are declared and executed.

Figure 2 shows the declaration of a simple dataflow. A dataflow in Xflow is a combination of data elements, which are either connected through the DOM hierarchy or via id references with the src attribute. Each data node defines a named value map declared through the node’s children. Those children come in two types: typed ValueElements, such as float or float3, defining individual, named fields and other data elements, whose value maps are simply merged into the parent’s map.

To process new values we can attach operators to data elements via the compute attribute. Operators are invoked just like regular JavaScript functions: we specify input parameters inside the parenthesis, referring data of the value map by name, and provide names for the output values to the left of the assigned operator. The output of the operator always overrides original values with the same name.

The declared dataflow is executed by requesting the named value map of a data element, which triggers the execution of all operators attached in the data element’s subtree, skipping those not relevant for the requested result. Before an operator of a data element is evaluated, usually all child data elements need to be evaluated first. However, note that Xflow does not simply use the hierarchy of data elements to determine the execution order but analyses the exact connection of operator in- and output, which often allows for more parallelization among hierarchically ordered data nodes.

3.2 Handling Image and Video Input

As Xflow is designed as a general purpose data processing tool, it can be applied to arbitrary data sets. In addition to (arrays of) integer, boolean, and float data types we support textures as input and output parameters. In this case data is provided by HTML img and video elements. Both img and video require special handling compared to other data types. Firstly, in order to start processing, we need the dimensions of input images to allocate the output accordingly. As the dimensions of images and videos are unknown until a certain part of the file is transferred, we need to suspend processing until they are available. Secondly, in contrast to img elements, a video is a stream of images. Thus, in order to process every frame, we register callbacks submitting each frame to the dataflow. These callbacks are usually invoked with a rate up to 60 times per second.

In addition, supporting video elements as input for Xflow allows for processing the video data stream produced by a webcam. We only need to activate access to the webcam for arbitrary video element, like shown in Figure 3. No further changes to the Xflow framework are required.

3.3 Operator Implementation

Now having as input an image or a video frame we can implement an image processing operator which is applied to the source image producing a new one. In Figure 4 we show an example of an operator implemented in JavaScript that computes a grayscaled image by setting red, green and blue components of the output image to the relative luminance \( Y = 0.2126R + 0.7152G + 0.0722B \) of the corresponding pixel of the input image.

Xflow allows us to register new JavaScript operators with the Xflow.registerOperator function. This function requires a name for the new operator and a declaration of input and output parameters. The output declaration allows us to specify the size of output images in several ways. The sizeof attribute creates an output image of the same size as an input image. When allocation of the output parameters requires more complex logic, a customAlloc attribute can be used. Xflow will then call a user-defined alloc function before executing the operator to receive the dimensions of the output parameters.

The output and input texture parameters are passed to the registered evaluate function. As a datatype for representing image we use
Augmented Reality uses image processing in a different way than previously discussed. Rather than producing new images, it applies computer vision algorithms on the input images to output transformations between the camera and objects in the video stream (e.g. markers or detected object features). Consequently, augmented reality is connected in a different way with the 3D scene than other image-processing operators.

In this section we will demonstrate how several small extensions allow us to connect augmented reality computations with XML3D in two ways: purely declarative and with scripting.

### 4.1 Connection to Scene Graph

An augmented reality operator, which places 3D objects relative to detected markers of a video stream, needs to influence three aspects in the scene graph: the transformation hierarchy of objects connected to the marker, the visibility of those objects and finally the projection matrix of the current view point to match the camera configuration of the video stream.

In order to allow Xflow to influence the transformation hierarchy of the XML3D scene graph, we had to extend the available transformations in XML3D. XML3D provides two ways to define the transformation of a group in the transformation hierarchy. The first approach allows the users to refer a `transform` element. The matrix is calculated from its set of attributes (translation, rotation, scale, center, etc.), similar to the transform node in OpenInventor and VRML [Web3D Consortium 1997]. The second approach is to define the matrix using CSS 3D Transforms [W3C 2012a]. Then the concatenation of several operators (e.g. `scale3d(value, value, value)`) is possible.
value), translate3d(value, value, value), etc.) defines the matrix. The concatenation corresponds to a matrix multiplication. The drawback of this approach is that manipulation is only possible by string manipulation which is inconvenient for complex animations.

We extended XML3D to allow a group to receive its transformation not only from a <transform> element, but also from a <data> element. The only requirement for the <data> element is to provide an output with name transformation and type float4x4. XML3D uses the first entry in this output and applies it to the transformation hierarchy. In combination with Xflow, this minimal extension provides a generic and flexible way to specify plain matrices, animations based on key frames in various formats, procedural animations and animations based on sensor input as we need it for AR applications.

Next, we need to modify the visibility of objects, which might need to be hidden if the corresponding marker is not detected. For this we chose a rather pragmatic approach of connecting the Xflow output to a custom shader, which takes a boolean parameter for visibility. No further extensions to the framework were required, as shader already support a connection with Xflow. We just had to add a custom shader, which is already possible with the current xml3d.js implementation.

Finally, in order to efficiently update the projection matrix of the current viewpoint to match the field of view of the video stream, we added a new perspective attribute to the <view> element, which links 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 the the declared content.

We implemented an augmented reality operator in Xflow using JSARToolkit internally. With the extensions described in this section, we created an augmented reality applications purely based on declarative content. See Figure 7 for an overview of the declarative connection of AR with the 3D scene graph.

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 of each visible marker), they should not slow down the application significantly.

In order to integrate scripts inside an XML3D and Xflow application, we implement the XML3DDataObserver. This observer is designed similar to the MutationObserver [W3C 2012b] and allows applications to observe complete or subsets of 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 MutationObserves, 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 8. 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 Automatic threshold computation

The marker detection algorithm of JSARToolkit first creates a binary version of the input image according to a given threshold value. The threshold value is usually determined automatically, making the marker detection algorithm less vulnerable to lighting conditions. However, this automatic threshold computation is not provided by JSARToolkit or at the very least not documented.

In order to demonstrate the capabilities of our approach we combined thresholding with Otsu’s method [Otsu 1979] with our AR operator based on JSARToolkit. The Otsu’s method operates on the histogram produced from the grayscale image. Thus we compute a grayscale image first, then create the histogram and pass it to the

---

2https://github.com/xml3d/xml3d.js/wiki/Material-Shaders#wiki-Custom_shaders
threshold operator. Finally, the computed threshold is passed to our AR operator (see Figure 9).

This example shows that by using Xflow’s generic approach we can split many complex algorithms into simple reusable operators and combine them as needed.

5 Parallelization

In the previous sections we showed an elegant approach to integrate image processing and augmented reality into Declarative 3D. We implemented this set of functionality in xml3d.js, our poly-fill implementation of XML3D and Xflow, that is purely based on JavaScript and WebGL. Consequently, our initial implementation of the image processing operators is based on single-threaded JavaScript, which resulted in rather poor performance.

In this section, we describe how we use available and emerging Web technologies to parallelize the execution of our image processing operators in two ways: we speed up individual operator execution with data-parallelization and improve usability by executing image processing in parallel to Web page rendering.

5.1 Parallelize Operator Execution

Many image processing operators essentially are projections from single pixels in the source image to a pixel in the result image, potentially taking some neighboring pixels in the source image into account. Moreover, the computation of each pixel often is independent and thus could be performed in parallel. With Parallel JavaScript, a data-parallel programming API for JavaScript proposed by Intel Labs [Herhut et al. 2012], it is possible to express such parallelism in JavaScript. Using Intel’s River Trail prototype\(^3\), the API can be translated for efficient execution on multi-core CPUs and SIMD instruction sets. As we show in Section 6, this offers significant performance improvements.

The key element of the Parallel JavaScript API is a new data-type ParallelArray. Compared to JavaScript’s existing Array object, a ParallelArray object mainly differs in its support for multidimensional arrays and the provision of common parallel methods like map, reduce, scan, filter, and scatter. To illustrate the use of ParallelArray objects, Figure 10 provides a translation of the relevant parts of the grayscale operator from Figure 4 to the Parallel JavaScript API\(^4\). The actual workload, i.e., the computation of luminance values has not changed. However, the for-loop in the sequential implementation has been replaced by a call to the ParallelArray constructor. As first argument, the constructor expects the size of the resulting ParallelArray structure. The created array is populated using the second argument: A function that, given an index position as argument, returns the corresponding value in the result array. Note that, to enable parallel execution, Parallel JavaScript requires the passed function to be free of side-effects.

In the given example, the constructor iterates over the number of pixels. However, for each pixel, we have to compute 4 values: a red, green, blue and alpha component. When using the ParallelArray API, this can be encoded by returning a vector of 4 elements. The resulting ParallelArray object thus is a two-dimensional array.

Once the ParallelArray object has been constructed, we assign its contents to the data field of the result image. As images in JavaScript use a special encoding as typed arrays, we have to convert the data contained in the ParallelArray accordingly. The asBinary method performs the conversion of the ParallelArray object into a JavaScript typed array. To reduce conversion overheads, we pass a hint for the desired storage type as third argument to the constructor, so JS can use the appropriate internal representation.

5.2 Separate Processing from Web Page Rendering

Certain image processing operations, such as complex computations in frequency domain, are not necessarily designed for real-time applications. Executing these image operators inside regular JavaScript code will freeze the web page and user interface resulting in a poor user experience. To avoid this problem, we designed our implementation to execute all image processing operators in a separate Web Worker. This allows the execution of JavaScript in a separate thread that communicates with the main thread by passing messages. Since our Xflow implementation operates independent of the DOM on custom data structures, we could easily move the whole dataflow structure to the Web Worker. The main thread simply sends the DOM structure and consequent DOM modifications to the worker thread that creates and synchronizes the dataflow structures. The web worker will repeatedly evaluate any modified part of the dataflow and send a message to the main thread with the processed images (or other data).

6 Results

6.1 AR Applications

To show that our declarative approach simplifies development of AR applications we created two example Web applications with our AR operator (see Figure 1). The first simple example visible in the center image, exclusively uses a declarative approach and places 3D objects on markers controlled by the user. The second example in the left image registers an observer (described in Section 4.2) and reacts on changes to the list of visible markers and their position, allowing us to implement dynamic movements between visible markers. For composing virtual objects with the real video stream produced by a webcam we use the standard HTML

\[^3\]The prototype is available at http://github.com/RiverTrail/RiverTrail.

\[^4\]We use the interface of the proposed API [Hudson and Herhut 2013]. A translation to the API of the River Trail prototype is straightforward.
6.3 Performance

The main concern of our framework is to provide a tool for rapid development of image processing applications. In addition, applications based on our framework are portable as they run in any modern browser without installing any plug-in.

Another important aspect is performance. Even though our implementation is based on high-level JavaScript code, we still get good performance results due to the continuous optimizations of JavaScript engines. Some of these optimizations, e.g. JIT compilation of JavaScript code, work especially well to speed-up tight inner loops, which are excessively used in the array based computation model of Xflow.

6.2 Image Processing Library

For the image processing library, we have already implemented some well-known algorithms in the following seven operator categories: basic pixel-wise, blending, spatial filtering, morphology, histogram processing, padding & cropping, and frequency domain operations. Figure 11 shows an example document structure for a sample filtering process in frequency domain using some of the implemented image processing operators. For the complete list of the implemented operators and the relevant documentation reader can refer to our project wiki pages at GitHub. Figure 1 shows a Web page where a large number of operators have been applied to the same input image.

7 Conclusion and Future Work

In this paper, we showed how a minimal extension to XML3D and Xflow allows us to integrate image processing and augmented reality into a Declarative 3D technology using generic dataflow processing directly connected to a 3D scene graph. We described all required extensions in detail, discussed optimization strategies and finally showed how easily applications can be implemented based on this technology.

The focus of this paper is on a clean integration model with Declarative 3D graphics. For future work there are many topics left to improve performance and flexibility of the system.

Xflow [Klein et al. 2012] supports merging of multiple operators for improved performance and the execution of dataflow on the GPU for character animations. We plan to implement the same function-

5https://github.com/xml3d/xml3d.js/wiki/Xflow-image-processing-operator-list
alibity for image processing and augmented reality.

While our operators do support execution on different platforms (native JavaScript or RiverTrail), we currently provide operator implementations for each platform separately. Thus, another goal is to find a common description in JavaScript, that can be used for all platforms, potentially even GLSL shader code.

Growing support for WebGL and WebRTC on the mobile devices allows us to use Xflow-based image processing and AR applications on this platform as well. However missing support for the hardware-acceleration in mobile browsers and insufficient JavaScript performance prevent us currently from making Xflow as fast as on the desktop. We hope to achieve this goal in the near future.

Finally, we showed how augmented reality can be implemented and integrated with Declarative 3D. However, the actual dataflow for AR consists of exactly one, very specialized operator, which goes against the design principle of Xflow to provide small, generic operators. We plan to provide more detailed computer vision operators that can be combined into dataflows for augmented reality and thus provides much more flexibility.

8 Acknowledgement

The character models of Figure 1 were provided courtesy of Valve Software.

The research presented in this paper has been generously supported by the Intel Visual Computing Institute as well as the EU projects Verve and FI-CONTENT, which is part of the Future Internet PPP program.

References


<table>
<thead>
<tr>
<th>Xflow Operator</th>
<th>Time in ms</th>
<th>Sequential</th>
<th>Parallel</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution 3x3</td>
<td>236</td>
<td>52</td>
<td></td>
<td>4.5</td>
</tr>
<tr>
<td>Convolution 5x5</td>
<td>502</td>
<td>89</td>
<td></td>
<td>5.6</td>
</tr>
<tr>
<td>Convolution 7x7</td>
<td>880</td>
<td>141</td>
<td></td>
<td>6.2</td>
</tr>
<tr>
<td>Convolution 9x9</td>
<td>1420</td>
<td>218</td>
<td></td>
<td>6.5</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.