

3D Graphics - Level of HTML Integration

Level 0 - Plugin integration model

This is the classical integration model for 3D graphics in the web. The scene description is independent from the web page and has its own run-time. Mouse events are propagated through the plug-in interface. The plug-in defines a plug-in API that allows to communicate between the plug-in and the webpage. The expressiveness of this API varies largely between different implementations.

Common plug-in APIs that are provided by the browser implementations: NPAPI, ActiveX, Pepper API.

Disadvantages:

- Not all browsers support all APIs
- Security issues (on non-sandboxed plug-in systems)
- Poor acceptance of users
- Weak integration with web technologies
- High entry barrier (completely new runtime)

Advantages:

- Highest control for vendors (on non-sandboxed plug-in systems)
- Closed system (IP issues)

Common systems using the plugin integration model: Unity, X3D plug-ins (Bitmanagement BS Contact, Fraunhofer Instant Reality)

Level 1 - API

This integration model came with **WebGL**: The user can acquire a context to perform low-level GL call to from the HTMLCanvas element. The integration is limited to this element, which of course is in the DOM and can be styled using CSS. But apart from this, there is no integration with HTML or CSS. The data and structure is typically in TypedArrays that are feed to the low-level API. Most applications use a JavaScript scene graph library (e.g. Three.js) that provides a higher abstraction level.

Level 2 - DOM-Integration

This integration level goes far beyond level 1:

- The **DOM** is the scene graph (i.e. tree)
- The scene can be arbitrarily modified via the **DOM API**
- **Reuse** of existing **HTML elements** (e.g. as texture)
- Support of **DOM events** (e.g. extended MouseEvents) and event attributes
- Provide 3D specific **base types** (matrices etc) for JavaScript
- Make usage of **existing** CSS properties

This level of integration is defined by what is currently feasible to emulate with existing APIs and JavaScript or plug-ins (Polyfill). Most prominent Polyfill implementations are xml3d.js and X3DOM. Though the names suggest something different, both approaches define extensions to HTML5. Both systems implement the features above by:

- Adding interfaces to generic DOM elements
- Monitoring all changes on generic DOM elements in JavaScript
- Using WebGL or plug-in (X3DOM) to render the scene
- Dispatching extended mouse events on the generic DOM elements

The Polyfill implementation works already very good. They have a strong user base and a lot of industry interest. Nevertheless, there are issues that come with this approach:

- **CSS Monitoring**
Though we can use CSS3 3D Transform for the transformation hierarchy, there is no way for an efficient implementation. From JavaScript, we don't have a way to track CSS state changes. Thus we have to query the current state every frame and get a copy of the matrix. This is sufficient for small scenes but we run into performance issues on large scenes. The same is true for other CSS properties we want to use (e.g. opacity).
- **Efficient base types**
Currently, only CSSMatrix is a pre-defined base type we could use for the polyfill implementation. Unfortunately it's designed to create instances on every operation and integrates not very well with TypedArrays and WebGL [3].
- **Behaviour of generic DOM elements**
We have to extend generic DOM elements for our purpose. The behaviour of those generic elements (sometimes referred to as **HTMLUnknownElement**) inserted during parsing of unknown elements is very different across browser implementations.
- **Modification of DOM**
For the Polyfill implementation, we have to add an element to the DOM to render the content to (e.g. a canvas element for WebGL). This modification is not expected by the user. This could be solved by the proposed [Shadow DOM API](#).

This is just an excerpt of issues we are facing for the polyfill implementations. We are currently working on a more complete and explicit version. We think that many of the issues could be relevant for polyfills of other domains or for libraries.

Level 3 - Custom CSS and Debugging integration

This integration level is based on Level 2, but additionally:

- Provides a set of **3D-specific CSS properties**
- Has a deeper integration into **developer tools** (e.g. Chrome Developer Tools, or Firebug)

An example for this integration level is the experimental Chromium that implements XML3D. Here we can use the 'shader' CSS property to assign shaders to DOM elements. Also, it's possible to debug the scene in the developer tools: Clicking on a 3D object leads directly to the defining element. The 'shader' property is available in the CSS monitor and can be debugged and watched during runtime.

This level of integration is currently not achievable by a Polyfill implementation, because the required APIs are missing. It's not possible to introduce new CSS properties and it's not possible to extend developer tools by an API.

The CSS Issue can be solved by adding a CSS Module specific to 3D graphics. Since it is difficult to find the right set of CSS properties right away, it would be favourable to build prototypes based on dynamic CSS properties (similar to HTMLUnknownElement), that can be accessed through JavaScript. Note, that these properties can be ignored by the layout engine and don't need an optimized representation, as only the string values are accessed through `getComputedStyle()` and similar functions.

Level 4 - Full shader description via CSS

At this level of integration, the whole shader (material) is described using CSS. In modern graphics APIs, shader attributes are generic. This means, it must be possible to define shader attributes with arbitrary name and with a type.

Generally there are two approaches to achieve this:

1. Using the CSS value. This is done in [Filter Effects 1.0](#):

```
.shaded {  
  filter: custom(  
    url(distort.vs) url(tint.fs),  
    distortAmount 0.5, lightVector 1.0 1.0 0.0,  
    disp texture(disp.png)  
  );  
}
```

2. Using generic CSS properties (s. above)