Dynamic and interactive aspects of X3DOM

Conference Paper · June 2011

DOI: 10.1145/2010425.2010440 · Source: DBLP

Johannes Behr
Fraunhofer Institute for Computer Graphics Research IGD
72 PUBLICATIONS **1,794** CITATIONS
SEE PROFILE

Yvonne Jung
University of Applied Sciences Fulda
76 PUBLICATIONS **839** CITATIONS
SEE PROFILE

Dynamic and Interactive Aspects of X3DOM

Johannes Behr* Yvonne Jung† Timm Drevensek‡ Andreas Aderhold§

Fraunhofer IGD, Darmstadt, Germany

Abstract

The previous publications on X3DOM focused on the general integration model [Behr et al. 2009] and implementation strategies [Behr et al. 2010]. The aspects of dynamic and interactive worlds were an essential part, but not specifically addressed as such. The recent major additions to the system are CSS Animations and CSS 3D-Transforms as well as various forms of events for user interaction and system monitoring, which complement the existing design to support a large number of interactive and dynamic use cases. This overall design, including scene update mechanisms, animations, and a large number of DOM-based events are thus presented in this paper as part of a single overall system design.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality I.3.6 [Methodology and Techniques]: Standards—Languages

Keywords: X3D, HTML5, WebGL, DOM, Web Integration

1 Introduction

The goal of the X3DOM model [Behr et al. 2009; Behr et al. 2010] is an open and and human-readable 3D scene-graph embedded in the HTML DOM, which extends the well-known DOM interfaces only where necessary, and which thereby allows the application developer to access and manipulate the 3D content by only adding, removing or changing the DOM elements via standard DOM scripting – just as it is nowadays done with standard HTML elements like `<div>`, `<img>` or `<canvas>`. Thus, no specific plugins or plugin interfaces like the SAI [Web3DConsortium 2009] are needed. Furthermore, this seamless integration of 3D contents in the web browser integrates well with common web dynamic techniques such as DHTML and Ajax, and the web browsers already provide a complete deployment structure.

The integration model is on the one hand very similar to how SVG now is part of HTML, but on the other hand it has specific and often 3D related properties which are deeply situated in the field of material management and interactive systems. The material management, especially in relation to CSS, is still an open research topic, since user defined CSS properties are still not supported by HTML and any web browser. For interactive and dynamic systems a overall design can be presented which should be the foundation for further work in the field of scene updates, animation, and UI events.

*email:johannes.behr@igd.fraunhofer.de
†email:yvonne.jung@igd.fraunhofer.de
‡email:timm.drevensek@igd.fraunhofer.de
§email:andreas.aderhold@igd.fraunhofer.de

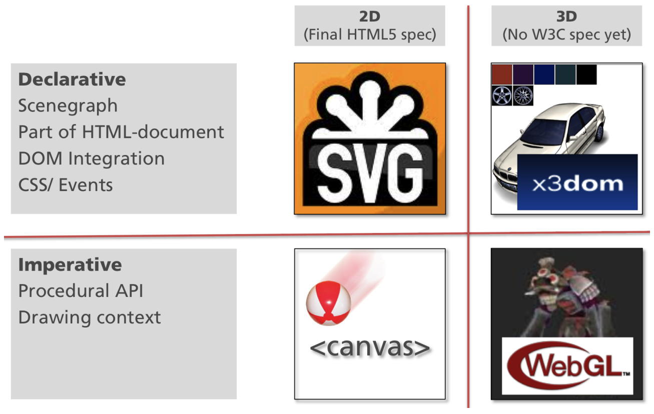

Figure 1: *SVG, Canvas, WebGL, and X3DOM relation.*

2 Related Work

The previous publications on the X3DOM framework focused on the general integration model [Behr et al. 2009] and implementation strategies [Behr et al. 2010], including X3D plugins and WebGL [Khronos 2011] backends. Dynamic and interactive aspects where mentioned, but not yet the main focus of these publications.

Lately, Khronos published the final WebGL [Khronos 2011] specification in March 2011, and there is already a large number of middleware components, which build on this low-level API [Kay 2010; Brunt 2010; Benedetto et al. 2010] to increase its usability. These libraries are comparable to typical graphics engines as well as to other JavaScript libraries like e.g. jQuery (cp. http://jquery.com/), but none of them seamlessly integrates the 3D content into the web page in a declarative way, nor do they directly connect the HTML DOM tree to the 3D content.

In addition, using libraries like SpiderGL [Benedetto et al. 2010] forces the web developer to learn new APIs as well as graphics concepts. But when considering that the Document Object Model (DOM) of a web page already is a declarative 2D scene-graph of the web page, it seems natural to directly utilize and extend the well-known DOM as scene-graph and API also for 3D content. Hence, the XML3D system [Sons et al. 2010] provides a very similar overall integration model, but in contrast to X3DOM is not based on X3D [Web3D 2008] at all, and the first paper here focused on the introduction of the new tag set, but likewise not particularly on dynamic and interactive aspects.

3 X3DOM Runtime System

Already the first paper on X3DOM [Behr et al. 2009] stated that the declarative 3D design includes not just a rendering abstraction but also a minimal X3D runtime system including e.g. animations and events. The design is therefore similar to how the SVG integration was designed (see Figure 1). It maps the DOM subtree to a scene-graph, which is part of the HTML document and fully integrated in the DOM, and works with CSS and DOM Events. Therefore it is not a standard X3D runtime [Web3D 2008] like the Instant Reality system [IR 2010], since we incorporate DOM changes and events as major form of update instead of X3D sensor nodes and the like.

The X3DOM system only contains a minimalistic X3D runtime and maintains the current state of the scene graph, redraws the scene as needed, receives input from the User Agent's (UA) event system and propagates field/ attribute changes though standard X3D ROUTES [Behr et al. 2009]. Moreover, the runtime system must be independent of any specific implementation or backend (e.g. browser plugin or WebGL), as already introduced in [Behr et al. 2010]. The runtime environment detects changes in the DOM and coordinates the processing of events, both are the primary means of generating behaviors in HTML and X3D. The overall system tries to minimize redraws and only draws the scene anew if the scene, viewport, and browser page is visible, and if some scene data has changed or should have changed according to CSS changes. The main categories for updates are therefore per-frame, time-based trigger, scene changes, user input, and viewport visibility.

3.1 Per-frame Updates

X3D and X3DOM, like most modern interactive graphics systems, do not depend on a fixed refresh rate. Therefore, per frame updates should not be triggered by an extra *setInterval()* or *setTimeout()* function. The application developer should provide a function instead, which will be called every animation frame automatically by the x3dom runtime:

```
x3dom.runtime.enterFrame = funcRef;
```

This general idea of a per-frame callback is not new at all, but really helpful, especially for smooth but efficient animation. Syntax and naming are borrowed from the WebGL [Khronos 2011] related proposal for *window.onbeforepaint* [Mozilla 2011a].

3.2 Time-based Updates

CSS Animation [W3C 2010] and X3D *TimeSensor* nodes [Web3D 2008] are both methods to provide time-based triggers and are mainly used for animation purposes. Both do not change the scene-graph directly, but fire an X3D event or update the CSS state. For more information see section 4.

3.3 Scene Changes

The scene-graph holds the declarative 3D description and all (direct or indirect) changes of the scene should trigger the redraw if necessary. This includes includes the following three main cases, where the first two where already explained in [Behr et al. 2010].

DOM element changes This includes the inserts/ removals of element and attribute changes. The application developer can use JavaScript standard methods like *elem.appendChild()*, *elem.removeChild()*, and *elem.setAttribute()*. The latter function needs text encoded values but can handle e.g. HTML colors like 'blue' directly.

Field interface Using the standard *setAttribute()* method to update attributes and therefore X3D field values every frame is quite inefficient for large updates (e.g. 10,000 vertices), since the data has to be encoded and parsed as text value. To accelerate accessing and updating large single- and especially multi-valued fields we support a subset of X3D's SAI standard interface to access the field values via *elem.getField()*.

CSS changes An update can also be triggered if any related CSS structure has changed.

It is essential to notice that the first two changes mentioned above are monitored and can lead to direct actions. The CSS state however has to be pulled frequently, since there is no notion of time or frame in the CSS subsystem.

3.4 User Input and Events

One of the main advantages of the DOM integration is the interactivity of the content, which can also easily be scripted by utilizing standard web APIs and architectures for integrating user interactions and 3D contents. Generally, the user interaction thereby can be classified into two main categories: navigating the virtual camera through the scene and interacting with 3D content (e.g. picking).

Navigation Steering and moving the viewpoint. This form of interaction changes the camera transformation parameters and therefore needs to trigger a redraw (see section 5).

HTML UI Events Interacting with the scene additionally can be achieved through UI events only, which thereafter can be used to change certain parts of the scene. Therefore, e.g. picking a box in 3D does not necessarily trigger a redraw (section 6).

Environmental Events These are events that build on HTML events, but which do not reflect a specific form of user interaction. They get fired e.g. if the transformation of an object has been changed or if a specific sub-tree is now visible. All those events again do not trigger a redraw automatically (compare section 7 for our proposal).

In this regard, it is important to notice that any given event type (e.g. HTML UI Event or the Environmental events) does not trigger a scene update or redraw automatically. This could be a follow-up, but it depends on user-code and application logic.

3.5 Viewport Visibility and Refresh Rate

The scene should only be redrawn if its viewport is visible, and the refresh rate should not exceed the screen refresh rate. Both would waste resources, which is especially important for mobile devices. To follow this basic and obvious rule is not always easy to implement but critical in multi-tasking environments. Visibility can be easily detected in a native client, and most plugin interfaces provide callbacks for visibility, but not for the screen refresh rate.

Until very recently, JavaScript didn't provide the ability to access this kind of information. Therefore, a new function called *requestAnimationFrame()* lately was introduced by Khronos [Mozilla 2011a] and there already exist various implementations in almost all UAs. The method now can be used instead of *setTimeout()* or *setInterval()*, and will only be called if the canvas is visible with the maximum refresh rate the display supports.

4 Animation and Automated Scene Changes

One goal of the system design is to include declarative animation and automated scene changes into the scene description. Basically, two possibilities are incorporated, which will be described here.

4.1 CSS 3D-Transforms & Animations

There are several possibilities to animate virtual 3D objects (e.g. for showing a device in action or visualizing a task), ranging from updating attributes in a script every frame over standard X3D *Interpolator* nodes [Web3D 2008] up to using CSS-3D-Transforms and CSS-Animations, which are currently given as W3C working draft [W3C 2010], and which are only implemented in WebKit-based web browsers, such as Apple Safari and Google Chrome.

While X3D interpolators are supported by current Digital Content Creation (DCC) tools (e.g., Blender) and are also able to animate vertex data (e.g. coordinates or colors), CSS animations are easily accessible using standard web techniques. However, despite their smooth web integration, they are only useful for simple rigid transformation changes. The following little code fragment shows an example on how to use CSS 3D-Transforms to update *Transform* nodes for animating their child nodes.

```
<style type="text/css">
 #trans {
    -webkit-animation: spin 8s infinite linear;
 }
 @-webkit-keyframes spin {
     from { -webkit-transform: rotateY(0); }
     to   { -webkit-transform: rotateY(360deg); }
 }
</style>
...
 <transform id="trans">
    <transform
     style="-webkit-transform: rotateY(45deg);">
    ...
   </transform>
 </transform>
```

4.2 X3D Interpolators & Followers

Like the CSS 3D-Transforms and Animations, the X3D standard also brings a well-defined animation framework with corresponding nodes, called interpolators [Web3D 2008]. The range of animation functionality by the two approaches nearly matches by their animation variety (linear, easing or Bezier). In extension to the data type support of the CSS transformations, the X3D interpolators are offering the same set of types plus interpolators for vertex-based keyframe animations like the *CoordinateInterpolator* or the *NormalInterpolator*, which allows morphing of meshes etc.

The main difference between interpolators and CSS animations is defined by the binding of interpolators to a value type and not to a destination property. Any interpolator acts as a single step inside an animation pipeline. This approach allows creating a signal chain for typed fields like vectors or scalars. While the interpolator itself acts as a generator, the output can be routed through any amount of pipeline elements like chaser or damper, which are part of the X3D followers component [Web3D 2008]. So the actual output to a transformation can be modified by a set of intermediate steps. Another advantage of X3D interpolators is their keyframe-based nature. While CSS animations are bound to single animation steps, interpolators can have unlimited keys and non-equidistant key times, which is typical for 3D animations.

X3D *Follower* nodes support the dynamic creation of parameter changes by receiving a destination value, e.g. given by a previously clicked 3D position. Upon this they create an animation that transitions the output value from the current value towards the newly set destination value by internally using the concept of linear filters as known from signal processing [Stocker 2006; Web3D 2008]. Like the interpolators, the follower concept is anchored in X3D by its field and route concept, and implemented in X3DOM.

5 Navigation

The rendering of virtual content inside a graphics pipeline ultimately is the orthographical- or perspective projection of 3D geometry onto a 2D plane. The parameters for this projection are usually defined by some form of virtual camera – here given as *Viewpoint* node. Any changes to the camera position or orientation, as well as changes at the perspective bias are leading to changes of the projection and view. The interactive manipulation of these camera parameters is usually called navigation.

5.1 Built-in Navigation

X3DOM, as a minimal X3D runtime system supports different built-in forms of navigation. This ensures that the application developer can easily provide an abstract definition without specifying a certain method. This gives the system the freedom to map a specific input device or device configuration (e.g. multi-touch screen) to a meaningful form of interaction. The basic types are set as the *NavigationInfo* node's "type" field.

The X3D standard [Web3D 2008] defines the following navigation types, which are almost self-explanatory: **any** (allows the user to switch freely between all available modes), **fly**, **walk**, **examine** (suitable for examine a single object from every direction while rotating and zooming), **lookAt** (allows the user to pick a point of interest which the system will focus on), and **none**. The latter type disables the navigation interaction by the user with the scene and is suitable for controlling the camera with some other technique.

The built-in navigation functionality thereby not only allows to quickly create dynamic 3D scenarios, but can also be more efficiently implemented internally. This is especially important in case picking is required like for the walk mode, which is implemented by rendering the scene from different directions (forwards, downwards) into an additional buffer to avoid expensive traversals.

5.2 Application-specific Navigation

The system also supports application-specific implementations of any form of navigation. Since the virtual camera is part of the DOM, it can be manipulated directly. The developer can grab an event or device state (e.g., accelerometer) to perform an update of the transformation and viewing parameters via *cam.setAttribute()*. If the application should only support this specific type of navigation, it is important to first switch off the built-in mode with `<NavigationInfo type='none'>` in order to avoid any unintended interference.

6 DOM Events

The DOM specification (Document Object Model) defines a standard for accessing HTML (Hypertext Markup Language) and well-formed XML documents. Anything found in an XML or HTML document can be accessed, changed, removed, or added programmatically by means of a tree structure defined by the DOM [W3C 2000a]. The DOM Level 2 specification includes a module defining a generic event system that describes event flow through a tree structure, and provides basic contextual information for each event, whereas in this section we mainly focus on UI events. The following classes of events are outlined in the DOM events specification [W3C 2000b, Section 1.1]:

UI Events Generated by user interactions through an external device like a mouse, keyboard, etc.

UI Logical Events Device independent events such as focus change or element triggering notifications.

Mutation Events Events fired by actions that modify the structure of the document (e.g. node insertion).

6.1 Event Flow

Events, once fired, traverse through the DOM tree in two directions [W3C 2000b]. First, the events are propagated down the tree to the target element, this is called the *capture phase*. Once they reached their target, the event enters the *target phase*. Then, during the bubble phase, the events are bubbled upwards the tree back to the root element (see Figure 2). Event propagation may be canceled, which prevents the default action being executed. It is also possible to capture the event before it reaches its target. This mechanism allows handling of the event by one of the targets' ancestors before its actual target is reached.

As illustrated in Figure 2, the X3DOM implementation follows the event capture and bubbling phases outlined previously. The X3D element and its child elements are treated like any other HTML element embedded in the HTML document and are thus interpreted as another valid part of that document. In order to allow user interactions with the 3D scene, the X3DOM runtime obviously reacts to certain UI events. Depending on the action desired, for example rotating an object in the scene, mouse dragging is recognized and the scene adapted and rendered accordingly. In contrast to X3D, where the event flow is propagated through an orthogonal route graph connected between in- and out-slots (shown in red in Figure 2), in X3DOM the original DOM event objects are persistent throughout all phases and then bubbled upwards the tree to allow parent elements to react to it as well. However, it should be further explored, in how far the synchronous event processing of X3D interacts and integrates with the asynchronous browser environment.

6.2 DOM Level 2 Events on X3DOM

The DOM specification outlines various different event types in the UI category, which are required to be supported by X3DOM in order to manipulate 3D scenes. The following UI events are specified in the DOM Level 2 Events module and likewise processed by the X3DOM environment [W3C 2000b]:

click When a pointing device is clicked over an element. The click event is defined as mousedown event immediately followed by a mouseup event over the same location. The following sequence of events are fired: mousedown, mouseup, click. This event is comparable to the X3D *TouchSensor*'s "touchTime" event out-slot.

mousedown Fired when a pointing device button is pressed over an element – comparable to "isActive" TRUE.

mouseup Fired when the pointing device button is released over an element – comparable to "isActive" FALSE.

mouseover Fired when the pointer is moved over an element – comparable to "isOver" TRUE.

mouseout Fired when the pointer is moved away from an element – comparable to "isOver" FALSE.

mousemove Fired when the pointing device is moved while over an element. This event is mostly comparable to "hitPoint_changed", whereas (as always for DOM events) the mouse position is an attribute of the fired Event object.

With the current DOM Level 2 specification there are no provisions for keyboard events. Therefore key events are considered non-standard. Most browsers vendors however implement non-standard keyboard events with their browsers:

keydown Fires when a key is pressed and repeats as long as the key keeps being pressed.

keypress Fires when a character is being inserted into an e.g. text input element. It repeats as long as the key is pressed (not recognized on WebKit/iOS).

keyup Fires when a key is released after the default action of that key has been performed.

These events are also used by X3DOM to allow for keyboard -based interaction on the X3D root element, whereas in the X3D standard key events are handled by the *KeySensor* node instead, which generates events when the user presses keys on the keyboard [Web3D 2008]. In order to capture for keyboard events, an HTML element needs to be able to acquire focus. Since focus behavior with X3DOM presents an opportunity for future research, it is not yet implemented on an per-element level. Hence, capturing keyboard events is currently only possible for the whole X3D scene.

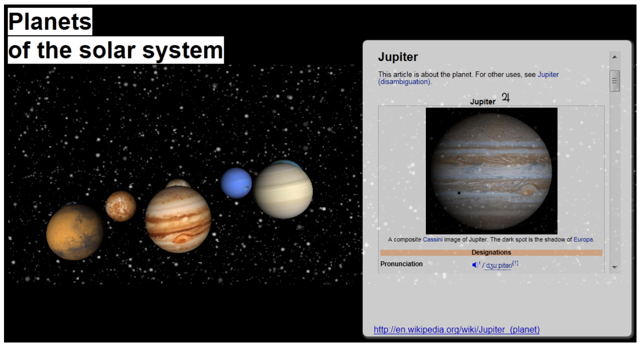
Planets
of the solar system
Jupiter
This article is about the planet. For other uses, see Jupiter (disambiguation).
Jupiter ♃
A composite Cassini image of Jupiter. The dark spot is the shadow of Europa.
Designations
Pronunciation
/ˈdʒuːpɪtər/[1]
http://en.wikipedia.org/wiki/Jupiter_(planet)

Figure 3: *Carousel menu: 3D navigation through 2D "mouse-move" event combined with 3D "click" event to choose a planet.*

To be able to deal with multiple X3D elements here, the corresponding internal event listeners are – like the mouse events – also attached directly to the internal canvas element, whose "tabindex" property therefore must be set to "0" to be focusable. During runtime, focus itself is given by clicking onto the whole X3D element.

In order to allow the application developer to hook into the event chain and perform custom actions in reaction to UI events, the X3DOM runtime allows to attach event handlers to the following X3D elements: *X3DBoundedObject* like *Group*, *Transform*, and *Shape*, and *X3DGeometryNode* like *IndexedTriangleSet* or *Box*. The following example, where the object's color is changed on click, illustrates how the developer can influence the behavior of the 3D scene by reacting to e.g. a click event on a *Shape* node.

```
<x3d>
  <scene>
    <shape id="shape">
      <appearance>
        <material id="mat" diffuseColor="red">
        </material>
      </appearance>
      <box></box>
    </shape>
  </scene>
</x3d>
...
changeColor = function() {
  var m = document.getElementById('mat');
  m.setAttribute('diffuseColor', 'green');
};
element = document.getElementById('shape');
element.addEventListener('click', changeColor);
```

The DOM specification also defines mutation events – events that are fired when the documents is being modified [W3C 2000b]. As already outlined in [Behr et al. 2010], some X3DOM backends (e.g. the WebGL-based one) currently watch and react to the *DOMNodeInserted*, *DOMNodeRemoved*, *DOMAttrModified* events, in order to allow the application developer to hook into the chain and perform custom actions.

6.3 3D Pick Events

Most visible HTML tags can react to mouse events, if an event handler was registered. The latter is implemented either by adding a handler function via `element.addEventListener()` or by

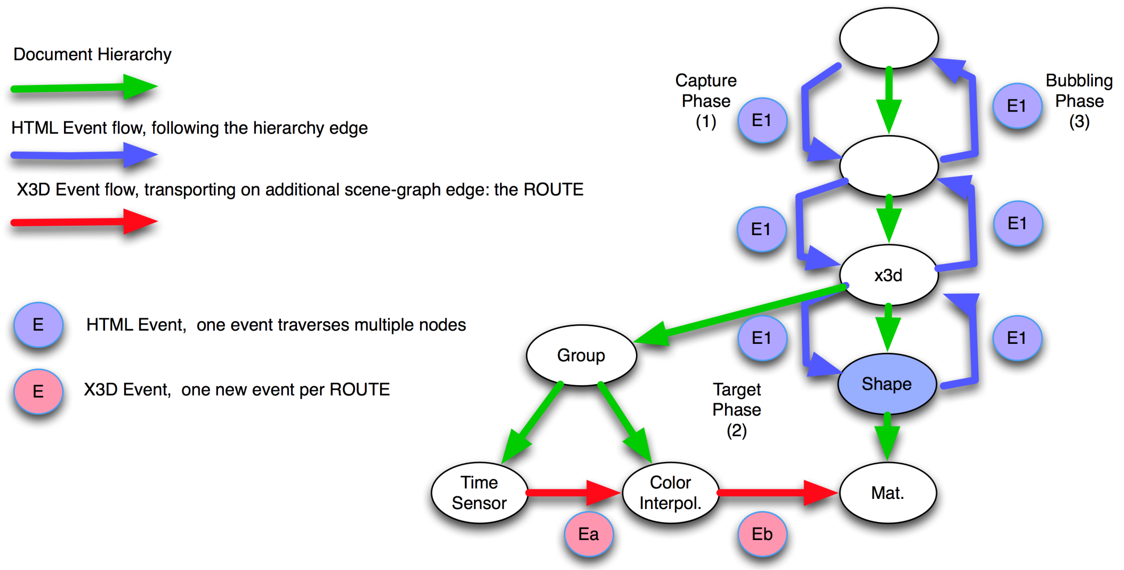
Document Hierarchy
HTML Event flow, following the hierarchy edge
X3D Event flow, transporting on additional scene-graph edge: the ROUTE
E
HTML Event, one event traverses multiple nodes
E
X3D Event, one new event per ROUTE
Capture Phase (1)
Bubbling Phase (3)
Target Phase (2)
E1
x3d
Group
Shape
Time Sensor
Color Interpol.
Mat.
Ea
Eb

Figure 2: *Event flow of DOM events (blue) vs. X3D events (red) that are propagated via special routes.*

directly assigning it to the attribute that denotes the event type, e.g. onclick. As already mentioned supports the X3DOM system standard HTML mouse events like *onclick*, *onmouseover*, or *onmousemove* are also supported for 3D objects alike. In addition, we also propose to create a new "3DPickEvent" type, which extends the W3C MouseEvent IDL interface [W3C 2000b] to better support 3D interaction. The new interface is defined like follows:

```
interface 3DPickEvent : MouseEvent {
  readonly attribute float worldX;
  readonly attribute float worldY;
  readonly attribute float worldZ;
  readonly attribute float localX;
  readonly attribute float localY;
  readonly attribute float localZ;
  readonly attribute float normalX;
  readonly attribute float normalY;
  readonly attribute float normalZ;
  readonly attribute float colorRed;
  readonly attribute float colorGreen;
  readonly attribute float colorBlue;
  readonly attribute float colorAlpha;
  readonly attribute float texCoordS;
  readonly attribute float texCoordT;
  readonly attribute float texCoordR;
  object getMeshPickData (in DOMString vertexProp);
};
```

This allows the developer to use the 2D attribute (e.g. screenX) and/ or the 3D attributes – e.g. worldX, worldY, worldZ or localX/ -Y/ -Z for 3D world or object local coordinates respectively –, if the vertex semantics are given appropriately (in this case the positions). Analogously, surface normal, texture coordinate, and color (if set) can be accessed via the correspondent attributes (e.g. normalZ). In general, the 2D/ 3D event now bubbles, as expected from standard HTML events, through the DOM tree and can be combined with e.g. a typical 2D event on the X3D element as in the application prototype shown in Figure 3.

Since all mouse and keyboard events are also supported directly on the `<x3d>` element, it allows the application developer to design own forms of navigation or mix 2D and 3D references freely. The example shown in Figure 3 utilizes such events to create a simple carousel menu, where the 3D navigation is achieved with user-defined JavaScript code through a 2D mouse event. Here, moving the mouse over the scene causes the planets to rotate. A standard mousemove event on the X3D root element is used to get the mouse pointer's position values to calculate the planet's rotation.

However, these events are not just useful to build own navigation types, but great for anything that needs pixels without a clear reference to a specific object like a context menu. If additionally an object reference is needed, one can also use these events on the 3D elements (e.g. `<group>` or `<shape>`) to get a standard MouseEvent, which provides pixels and additional 3D properties (e.g. worldX). Picking of the 3D objects, i.e. the planets, hence is achieved using a standard click event, where clicking on a planet reveals additional information (which in this example links to an appropriate Wikipedia entry) embedded in an overlay. Likewise, JavaScript's click event applied upon a planet's geometry is used as the trigger.

The `getMeshPickData()` method additionally can be used to access generic vertex data. Depending on the currently selected interaction mode, different actions can be performed for the scene, e.g. color or texture coordinate picking. Figure 4 (middle/ right) shows a first 3D CAE prototype for the visualization of simulation data in the web browser using X3DOM, where two examples from the domain of sheet metal forming (middle) and electromagnetic field simulation (right) are taken. These application prototypes have a huge potential to ease the communication and presentation of simulation results dramatically, without distributing a whole application while providing more information than a screenshot.

Here, the example in the middle shows the results of a sheet metal forming simulation. The material thickness after the forming process is color coded and applied via a look-up texture. By clicking onto a colored region, the corresponding thickness value is obtained and marked with a yellow arrow and textual information in the top right of the application. The sliders below can be used to interactively modify the color coding by setting offset, bias, and threshold. The leftmost image in Figure 4 shows the visualization of a flow simulation, where the respective color can be retrieved by picking a certain 3D flow region. This color again maps to a specific temperature, which is obtained by inverting the original look-up value. An in-depth description of techniques for scientific visualization for instance can be found in [Schumann and Müller 2000].

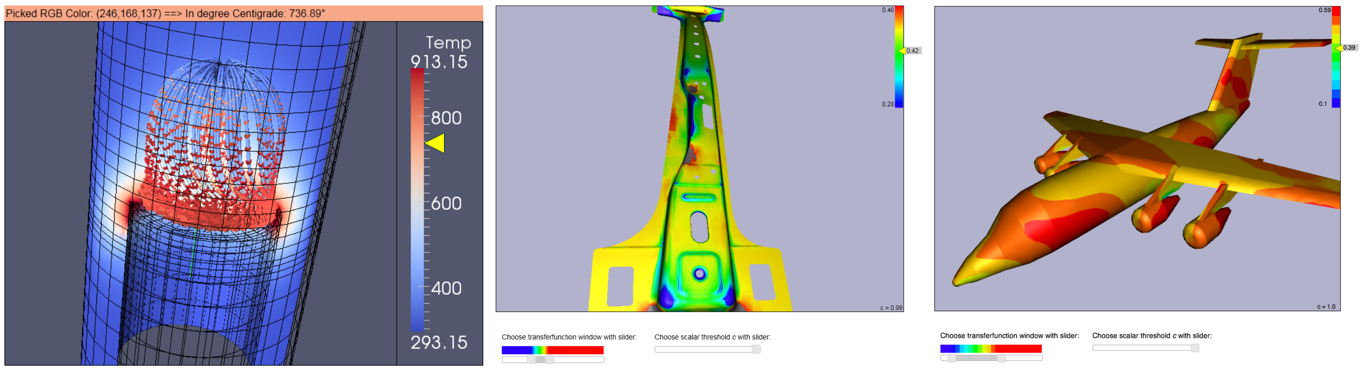
Picked RGB Color: (246,168,137) ==> In degree Centigrade: 736.89°
Temp
913.15
800
600
400
293.15
Choose transferfunction window with slider:
Choose scalar threshold c with slider:
Choose transferfunction window with slider:
Choose scalar threshold c with slider:

Figure 4: *Picking of generic vertex and/or fragment properties (left: temperature, middle: material thickness, right: field strength).*

6.4 Touch and Multitouch Events

With the recent proliferation of touch and even multitouch devices, corresponding interfaces become also of interest in browser APIs. However, the current DOM specification is still lacking a touch events module. Effort is being taken by major browsers vendors to implement this functionality as non-standard extensions [Mozilla 2011b].[1] Apple's proprietary Webkit/iOS implementation basically provides a feature-rich reference of how such functionality could be integrated into browsers [Apple 2010, Chapter 6, p.59 ff].

In the context of X3D, in [Jung et al. 2008] basic multitouch functionality was proposed with the *HypersurfaceSensor* node, a specialized pointing device sensor that handles typical multitouch interaction. However, web technologies and the DOM already provide an established path for handling user interaction, whereas the X3D sensor concept seems rather odd to web developers. The explicit goal of the X3DOM runtime is to minimize duplication and to use established standard technologies in the web application domain. A developer familiar with DOM scripting is able to program touch interaction using X3DOM by means of registering event handlers for touch events.

Although touch events are not yet widely supported by browsers, some vendors have entered experimental stage and with Apple already providing a production ready solution on Webkit/iOS, other vendors are following. Therefore we propose to extend the X3DOM system with multitouch support by choosing Apple's API as template. Such extensions would result in the following events that could be registered with X3D nodes (however, the device driver therefore needs to support touch and gesture events concurrently):

touchstart Fired when an object is placed on the screen.
touchmove Fired when an object (e.g. a finger) touches the device and moves over its surface.
touchend Fires when an object is removed from the screen.
touchcancel Fires when the touch action is aborted by the system.
gesturestart Fires when two or more objects are touching.
gesturechange Fires when two or more objects are touching the screen and moving.
gestureend Fires when two or more objects are not touching the screen anymore.

When one of the above events is fired, a *TouchEvent* object is passed into the function handling the event. The event object contains three lists: *touches*, *targetTouches* and *changedTouches*, which contain *Touch* objects with information about the currently active "fingers" including data like a unique identifier, the position on screen, and the event target.

We propose the introduction of a *3DTouch* object which amends touches with 3D information. In contrast to the *3DPickEvent* a *3DTouch* object is not an event, but an object encapsulated in a list stored within a *TouchEvent* provided by the browser. This additional layer is required to track multiple touches per event. The proposed interface could extend the *Touch* definition specified in [Apple 2008] adding additional 3D information like shown next:

```
interface 3DTouch : Touch {
  readonly attribute float worldX;
  readonly attribute float worldY;
  readonly attribute float worldZ;
  ...
};
```

Similar to the 3DPickEvent outlined in section 6.3, the application developer can use 2D and 3D attributes to program interactions with the scene like in the following example.

```
<shape>
 <appearance>
   <material id="mat" diffuseColor="red"></material>
 </appearance>
 <box ontouchstart="document.getElementById('mat')
        .setAttribute('diffuseColor', 'green');"
      ontouchend="document.getElementById('mat')
        .setAttribute('diffuseColor', 'red');>
 </box>
</shape>
```

Short of Apple's iOS implementation, only the Gecko 2.0 engine currently provides experimental DOM support for touch events [Mozilla 2011b]. Multitouch and gestures are not supported either, leaving the implementation as future work for the X3DOM engine.

7 Environmental Events

Inspired by the X3D environmental sensors, which are nodes that emit events based on some other event that occurs within the environment [Web3D 2008], we propose three additional 3D events that extend the Dom Level 3 UIEvent. These can not only be triggered by the user, but also indirectly by an animation or an interaction between two elements within the world. The event interfaces are shown next.

```
interface 3DVisibilityEvent : UIEvent {
  readonly attribute float sizeHint;
  ...
}
```

[1]http://code.google.com/p/chromium/issues/detail?id=16305
http://www.chromium.org/user-experience/multitouch#TOC-Proposed-behaviors

```
interface 3DProximityEvent : UIEvent {
  readonly attribute object position;
  readonly attribute object orientation;
  ...
}

interface 3DTransformEvent : Event {
  readonly attribute array transformations;
  ...
}
```

The 3DVisibilityEvent fires whenever the visibility of the given sub-trees changes, like for example a planet that moves off the viewport area. The similar X3D concept is the *VisibilitySensor*, which likewise detects when a user can see a specific object or region [Web3D 2008]. Comparable to the X3D *ProximitySensor* that generates events when the viewer enters, exits and moves within a region of space that is defined by a box, the 3DProximityEvent is a sensor for detecting object-camera transformations.

Finally, the 3DTransformEvent fires in case of object transformations, and hence can be used similar to the out-slots of an X3D *Transform* node. This is done by attaching a callback to the `<transform>` element via *addEventListener()*, which listens for an event of that type. Obviously, most eventOut slots of X3D nodes, like the "position" of the *Viewpoint* node etc., can serve as basis for new 3D events, but this discussion is left open for future work.

8 Conclusions

Important factors for authoring and rapid development of web applications are the possibility for declarative content description, flexible content in general, and interoperability – i.e., write once, run anywhere (web/ desktop/ mobile). In X3DOM this is achieved by utilizing the well-known JavaScript and DOM infrastructure of web browsers also for 3D in order to bring together both, open architectures and declarative content design known from web design as well as imperative approaches known from game engine development. Moreover, the application-independent visualization enables context sensitive and on-demand information retrieval, which is even more of interest for distributed content development using available web standards. Thus, the unification of 2D and 3D media development is another essential aspect.

Therefore, in this paper the aspects of dynamic and interactive worlds were addressed. The recent major additions to the system are CSS Animations and CSS 3D-Transforms [W3C 2010] as well as various forms of events for user interaction and system monitoring, which complement the existing design of X3DOM to support a large number of interactive and dynamic use cases. We therefore presented the overall design, including scene update mechanisms, animations, and the incorporation of a large number of DOM-based events as part of an overall system design.

Future work will be mainly influenced by the discussions and results of the newly founded W3C incubator group "Declarative 3D". Furthermore, we'd like to explore in how far new HTML5 elements like the `<device>` tag can be utilized towards interactive Augmented Reality applications and so on.

Acknowledgements

Part of this research has been carried out within the Software-Cluster (EMERGENT), which was funded by the German ministry for education and research (BMBF), and within the V-MusT.net Network of Excellence that has received funding from the European Community's Seventh Framework Programme.

References

APPLE, 2008. Touch event interface definition. http://www.opensource.apple.com/source/WebCore/WebCore-351.9/dom/Touch.idl.

APPLE, 2010. Safari web content guide. http://developer.apple.com/library/safari/documentation/Apple-Applications/Reference/SafariWebContent/.

BEHR, J., ESCHLER, P., JUNG, Y., AND ZÖLLNER, M. 2009. X3DOM – a DOM-based HTML5/ X3D integration model. In *Proceedings Web3D '09*, ACM Press, New York, USA, 127–135.

BEHR, J., JUNG, Y., KEIL, J., DREVENSEK, T., ESCHLER, P., ZÖLLNER, M., AND FELLNER, D. 2010. A scalable architecture for the HTML5/ X3D integration model X3DOM. In *Proc. Web3D 2010*, ACM Press, New York, USA, 185–193.

BENEDETTO, M. D., PONCHIO, F., GANOVELLI, F., AND SCOPIGNO, R. 2010. Spidergl: a javascript 3d graphics library for next-generation www. In *Proc. Web3D 2010*, ACM, New York, USA, 165–174.

BRUNT, P., 2010. Glge. http://www.glge.org/.

IR, 2010. Instant Reality. http://www.instantreality.org/.

JUNG, Y., KEIL, J., BEHR, J., WEBEL, S., ZÖLLNER, M., ENGELKE, T., WUEST, H., AND BECKER, M. 2008. Adapting X3D for multi-touch environments. In *Proceedings Web3D 2008*, ACM Press, New York, USA, 27–30.

KAY, L., 2010. Scenejs. http://www.scenejs.org/.

KHRONOS, 2011. Webgl specification. https://cvs.khronos.org/svn/repos/registry/trunk/public/webgl/doc/spec/WebGL-spec.html.

MOZILLA, 2011. Mozbeforepaint. https://developer.mozilla.org/en/DOM/window.onmozbeforepaint.

MOZILLA, 2011. Touch events. https://developer.mozilla.org/en/DOM/Touch_events.

SCHUMANN, H., AND MÜLLER, W. 2000. *Visualisierung – Grundlagen und allgemeine Methoden*. Springer Verlag.

SONS, K., KLEIN, F., RUBINSTEIN, D., BYELOZYOROV, S., AND SLUSALLEK, P. 2010. Xml3d: interactive 3d graphics for the web. In *Proc. Web3D 2010*, ACM, New York, USA, 175–184.

STOCKER, H. 2006. Linear filters: animating objects in a flexible and pleasing way. In *Proceedings of the eleventh international conference on 3D web technology*, ACM, New York, NY, USA, Web3D '06, 119–129.

W3C, 2000. Document object model (dom) level 2 core specification. http://www.w3.org/TR/DOM-Level-2-Core/.

W3C, 2000. Document object model (dom) level 2 specification events module. http://www.w3.org/TR/DOM-Level-2-Events/events.html.

W3C, 2010. Css 3d transforms. http://dev.w3.org/csswg/css3-3d-transforms/.

WEB3D. 2008. *X3D*. http://www.web3d.org/x3d/specifications/.

WEB3DCONSORTIUM, 2009. Scene access interface(sai), iso/iec 19775-2.2:2009. http://www.web3d.org/x3d/specifications/ISO-IEC-FDIS-19775-2.2-X3D-SceneAccessInterface/.