ARML 2 – prototype implementation

Abstract

This document describes a stab at implementing some of the ARML2 draft standard, as per the document located at https://portal.opengeospatial.org/files/?artifact_id=52739. The current java-script based implementation works on the client side to synthesize appropriate DOM elements for the ARML2 constructs, that are then added to a Three.JS scene-graph and rendered using the CSS3DRenderer of Three.JS.

The code for the current implementation resides at https://github.com/indivisibleatom/arml2ImplProto.

Code-flow

Fig. 1 outlines the interaction between the different components of the system.

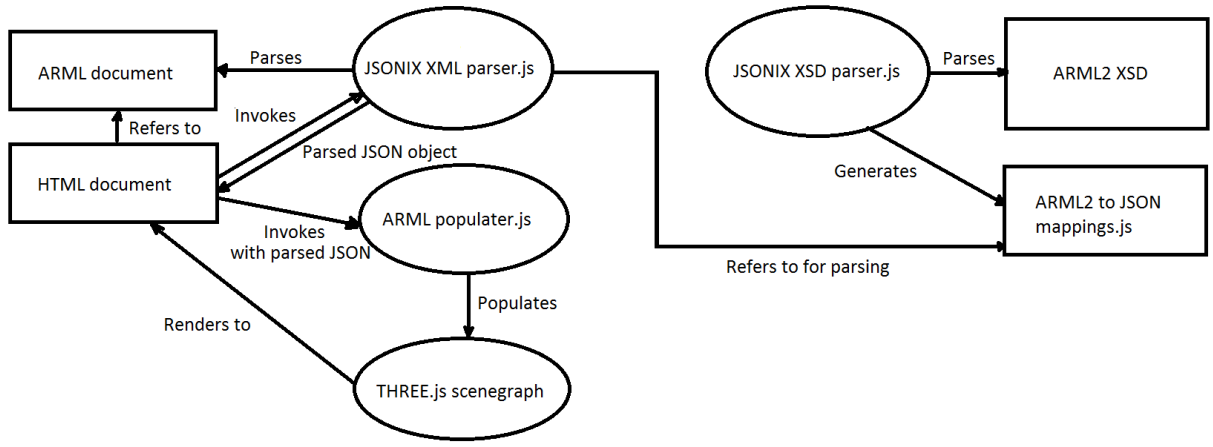

Fig. 1: Different components and their interactions in the system

The first step is offline and involves the creation of the JSON mappings for the ARML2 schema from the schema definitions (XSD). The XSD of the ARML2 schema is at present gleamed from the ARML2 draft specification document and is at armlMappings/armlXsd.xml. This is then followed by the usage of the JSONIX XSD parser to create a file containing JSON mappings for the XSD. The generated JSON mappings contain structures for the various types defined in the XSD and will later be used by the JSONIX XML parser to synthesize a JSON representation of a provided ARML2 XML file. The generated schema is located at armlCode/armlJSONSchema.js, and should be modified to be kept in sync with the latest ARML XSD. The readme file at armlMappings/readme.txt provides the syntax of the command to be executed for synthesizing the JSON mappings from a given schema file.

The subsequent steps all take place when a HTML page that is required to display ARML is requested by the client and run via JavaScript in the client's browser. A HTML page is required to initiate the parsing of the ARML document on the desktop mockup, without tampering with the ARML file itself to add the required JavaScript code. When translating over to ARGON, this extra level of indirection can be done

away with and the boiler-plate code used for synthesizing the HTML DOM from an ARML file can be moved into ARGON.

A sample HTML page that loads a sample ARML file is at armlCode/armlDemo.html, with the onLoad function being the hook that initiates loading of the ARML file when the HTML page load is complete. Note that this page is also augmented with a function to capture key-presses – this is simply to simulate geo-location changes, and will not be required when ARGON is used, as that provides explicit GPS based geo-locations, with bearings localized to the camera view frustum.

The onLoad function uses JSONIX's XML parser and the offline generated JSON mappings to populate a JSON object indicative of the ARML being loaded (this the prototype case, it is armlCode/sample.arml. This populated is then passed to the populateARMLType function of armlCode/armlPopulator.js.

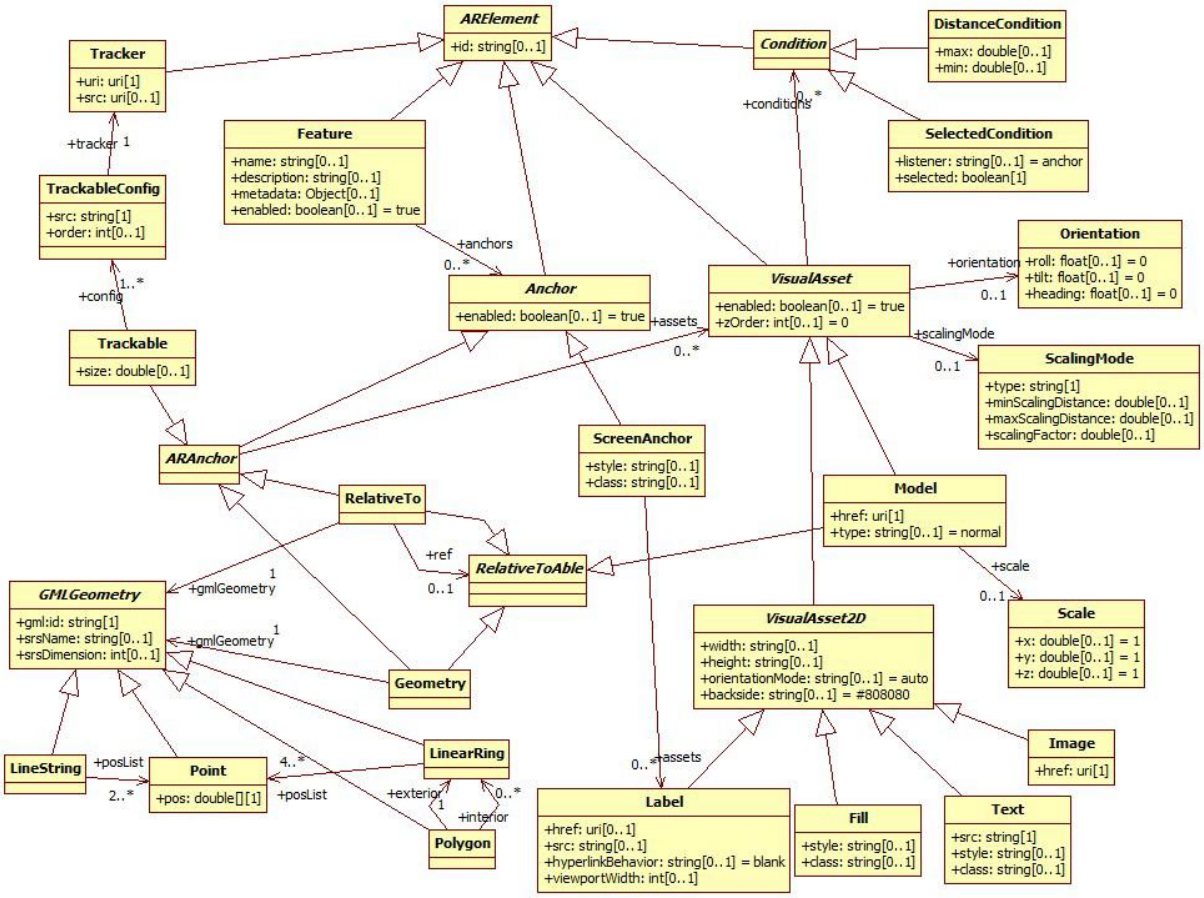

Fig. 2: ARML class hierarchy as obtained from the ARML2 specification document

The armlPopulator.js file is essentially an adaptor, which uses prototypal inheritance to mimic the ARML2 spec's class hierarchy (Fig. 2). Partially developed, it consists of creation function for a subset of the above types. The creation functions take in the relevant JSON object corresponding to their definitions as synthesized by JSONIX's XML parser, and use the information there to populate properties

that allow for easier handling by the AR application. Additionally, each of these objects have a cssElement function that synthesizes a DOM element from their contents.

For example, Fig. 3 show what the code for the Label class looks like:

```
Label.prototype = Object.create( VisualAsset2D.prototype );
function Label(labelType)
{
  VisualAsset2D.call(this, labelType);

  this.href = initOrSetDefault( "href", labelType, null );
  this.href = (this.href !== null) ? initOrSetDefault( "href", labelType.href, null ) : null;

  var srcContent = initOrSetDefault( "src", labelType, "" );
  this.src = "";
  if ( ! (srcContent === "") )
  {
    srcContent.content.forEach( function( srcContent ) {
      this.src += hasProperty( "outerHTML", srcContent ) ? srcContent.outerHTML : srcContent;
    }, this );
  }
  this.hyperlinkBehavior = initOrSetDefault( "hyperlinkBehavior", labelType, "blank" );
  this.viewportWidth = initOrSetDefault( "viewportWidth", labelType, 256 );

  this.cssElement = function() {
      var div = document.createElement( "div" );
      div.enabled = this.enabled;
      div.style += "z-index:" + this.zOrder + "; width:" + this.width + ";";
      div.innerHTML = this.src;
      return div;
  }
}
```

Fig. 3: ARMLpopulator code for the Label adaptor object

The constructor of the type uses the passed in JSONIX generated JSON representation of the label's XML to populate more intuitive properties such as href, srcContent, etc. Additionally, it has a cssElement function that returns a div DOM element, appropriately populated with the content mentioned in the label's description in ARML.

The ARMLpopulator generated structures are in a good format for rendering via THREE.js using its CSS3DRenderer. We traverse each Feature in the ARML description of the AR scene, and add the cssElement of each of the referred/contained anchors (and thus their referred assets) to the THREE.js scene's scene-graph.

Results

The current results demonstrate two features:

ScreenAnchor's synthesizing appropriate divs for rendering static content directly onto the documents DOM, and ImageAnchors being added to the CSS3DRenderers scene to allow for the scaling and rotation operations as the simulated geo-location moves near/far to/from the point of interest (Fig. 4).

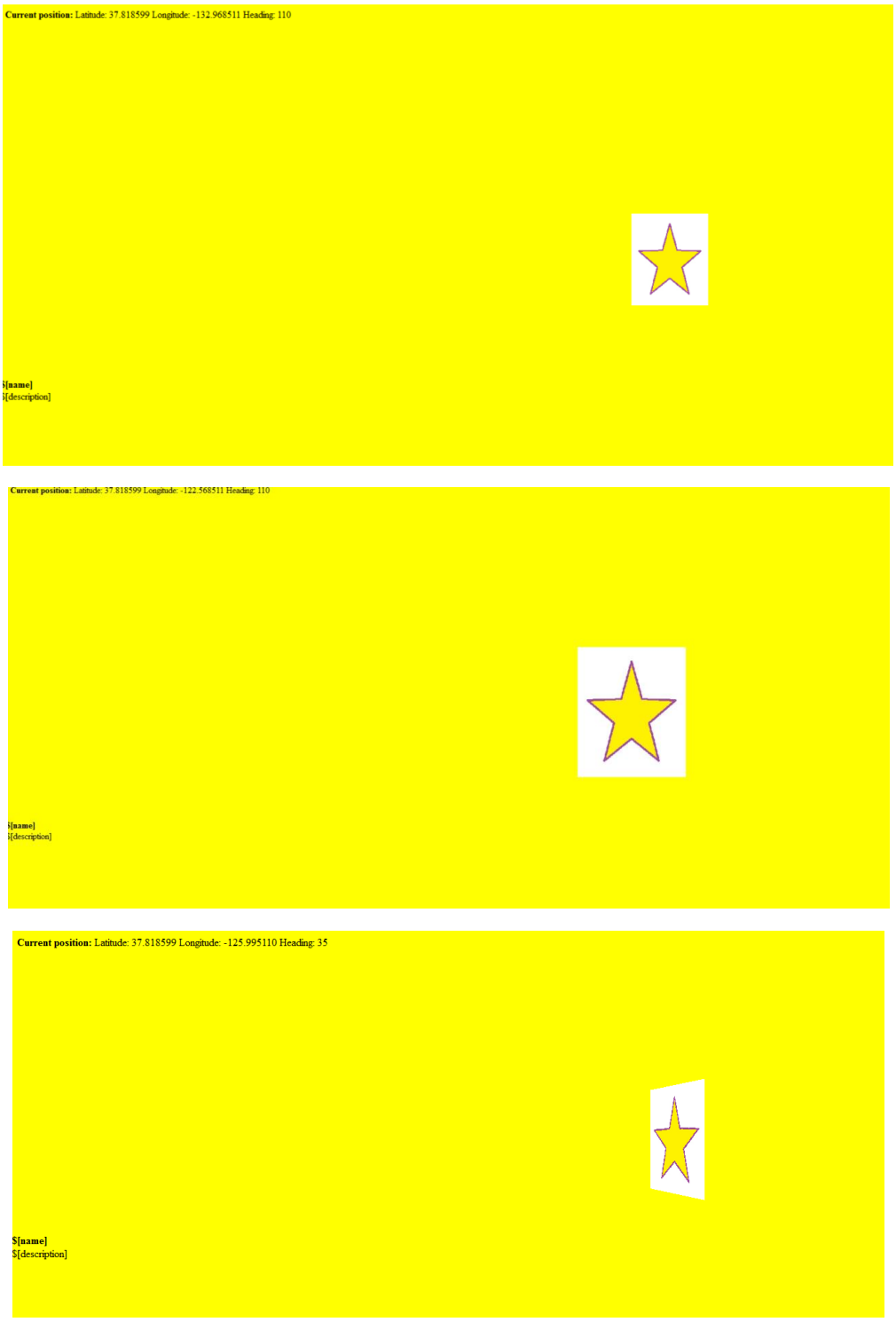

Fig. 4: Rendering of the ARML with POI dependent ImageAnchor scaling and rotations and fixed ScreenAnchors