SD226781

# Advanced Techniques for Using the Forge Viewer

Kevin Vandecar
Denis Grigor
Autodesk Inc.

## Learning Objectives

- Learn what Forge Viewer extensions are and how to load them;
- See the WebVR extension in action and understand how it can be used;
- Learn how to add custom geometry to a scene and how to change the appearance of existing model components;
- Learn how to manipulate component transformations and add custom interactions and animation.

## Description

The Forge Viewer is built on the three.js library and can be easily extended and mixed with three.js functionality. This class will demonstrate just how flexible and customizable the Forge Viewer is by walking you through a set of code samples. Topics covered will include WebVR support, mixing your model components with custom graphics, changing the appearance of your components by adding custom materials and shaders, and animating your models.

## Your Forge DevCon Expert(s)

***Kevin Vandecar:***

Kevin Vandecar is a Forge developer advocate and also the manager for the Media & Entertainment and Manufacturing Autodesk Developer Network API Support workgroups. His current specialty is 3ds Max software customization and programming areas. In recent years, h, he has also been working on his web development skills and exploring areas in WebGL and Three.js, along with the Autodesk Forge APIs.

***Denis Grigor:***

Within Autodesk, Denis Grigor is providing programming support to 3ds Max and Forge external developers. This mix of desktop and cloud exposure, helps him in his endeavor to safely erase the borders between them.
He likes to know how everything works under the hood, and he is not afraid of low-level stuff like bits, buffers, pointers, stack, heap, threads, shaders and of course Math.
Add to this his interest of IoT and IIoT along with love for 3D graphics and here you have a cyber alchemist.

The Forge platform https://forge.autodesk.com/ has been around for several years now, and at its heart is the Forge Viewer. The Viewer is a client-side JavaScript library based on three.js. In this class we will concentrate on enhancing the Forge Viewer behavior to provide more robust functionality.

The Forge Viewer is an important component to the Forge workflow. After a model has been translated to the Forge SVF format (see Mode Derivative API) it can be visualized and explored in a web browser, while keeping all the information related to the model (like component hierarchy, materials, properties and other metadata). Before the Forge Viewer technology became available, there were other, heavier options for viewing. Many were desktop, and required anyone viewing the model, to have software pre-installed. Even in the browser world, this was difficult without plugins until WebGL and HTML 5 came along making it possible to render 3D directly in the browser. Today most browsers and devices support WebGL and HTML 5, making viewing as simple as navigating to a website.

The Forge Viewer technology can also be found in A360, Fusion360 and more recently in Revit, 3ds Max, etc. as Shared Views.

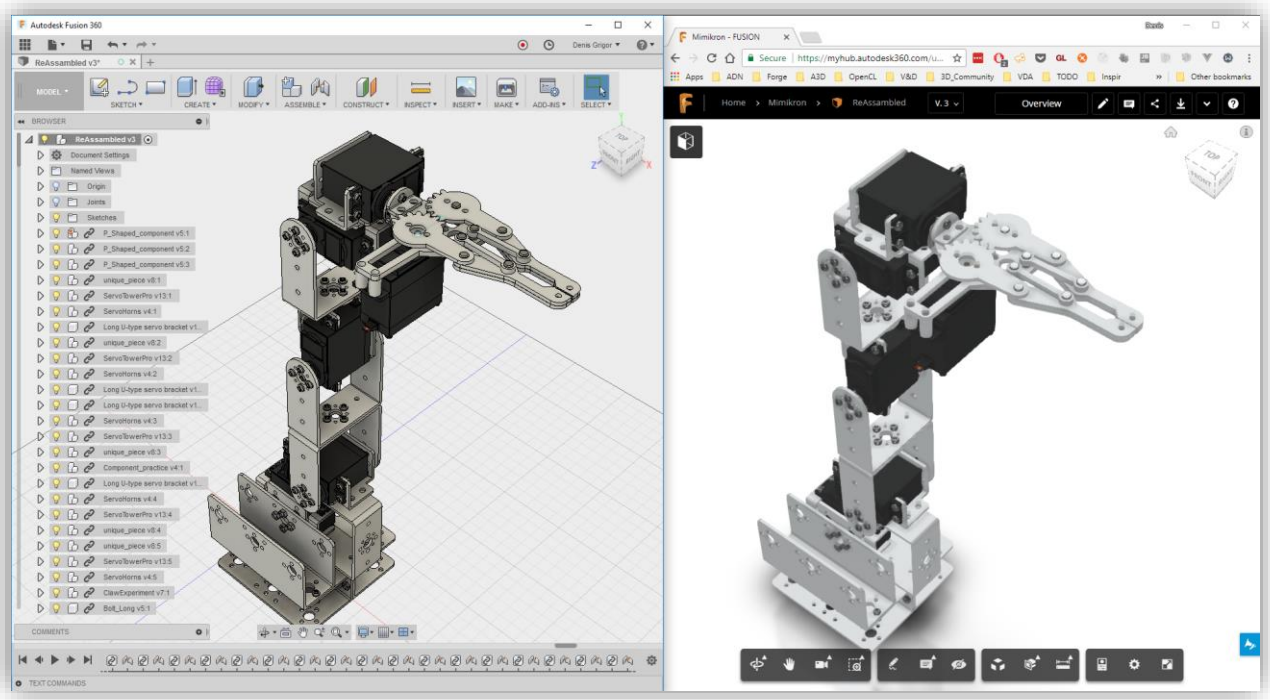


Figure 1

In Figure 1 you can see a model being displayed in Fusion 360 editor on the left, and the Fusion360 gallery showing the same model on the right for purposes of sharing and viewing the model.

## Forge Viewer for Developers

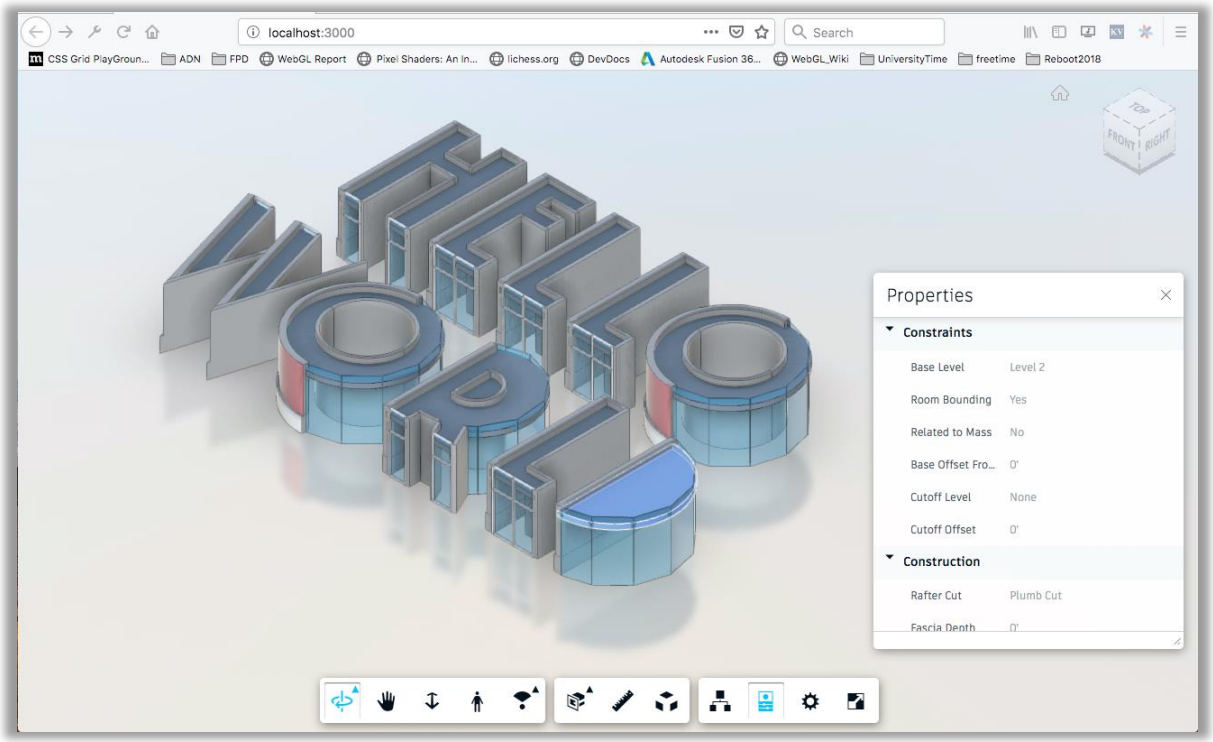
The Forge Viewer is a simple JavaScript library and as such can be configured on a webpage/webapp by referencing the JavaScript library, and then pointing to a Model Derivative SVF file. It provides 3D navigation in a consistent manner. It can be loaded with or without a standard toolbar and UI, making it very flexible. As a developer, the amount of code required to provide this viewing ability is a manner of a few lines of code, with a simple HTML 5 <DIV> tag.

Even though this is an Advanced class, it is important to understand the basics. The first step is to bring the model into a web page.

Here is the minimal code (providing the required URN and an access token):

```
<!-- The Viewer JS -->
<script src="https://developer.api.autodesk.com/modelderivative/v2/viewers/three.min.js"></script>
<script src="https://developer.api.autodesk.com/modelderivative/v2/viewers/viewer3D.min.js"></script>

<!-- Developer JS -->
<script>
    let viewer;
    let options = {
        env: 'AutodeskProduction',
        accessToken: '<YOUR_APPLICATION_TOKEN>'
    };
    let documentId = 'urn:<YOUR_URN_ID>';
    Autodesk.Viewing.Initializer(options, function onInitialized(){
        Autodesk.Viewing.Document.load(documentId, onDocumentLoadSuccess, onDocumentLoadFailure);
    });
    function onDocumentLoadSuccess(doc) {}

</script>
```

This is enough code to allow viewing of the model. For example:



As you can see, the original hierarchical structure and the metadata associated to each component was preserved and is available to the default instance of the Viewer. The combination of the Model Derivative service (which translates the model form the source into the SVF format) and the Forge Viewer provides a very powerful experience with minimal effort. Because this is developer technology, it can also be customized to provide unique use-case experiences. The main benefit compared with other in-browser viewing technologies is this exact preservation of the data. This technology is also designed to handle very large models with good viewing performance. In fact, on the user-side, this viewing technology is often referred to as the Large Model Viewer (LMV).

In the above example (a Revit model), when you click on a window, you are not presented with info on vertices, triangles etc., but information like height, width, type of materials and many other relevant in-context information. This is what makes it a professional grade viewer for many design industries.

With this minimal code, we have hosted our model viewable in a web browser, and it comes with integrated and standard navigation controls, camera control (perspective/orthographic), component tree, measuring tools, property tools and many other characteristics.

## Customize the Forge Viewer through extensions

Because the Forge Viewer is just a JavaScript library built on top of three.js, the possibilities of customization become vast. Although you can do things within the viewer instance that can be customized directly, there is one important feature that helps you to manage this customization into modules that can be loaded/unload at runtime. This functionality is called an **extension**.

The Forge Viewer concept of "**extension**" provides "a mechanism to write custom code that interacts with the viewer" and the official documentation provides a step-by-step tutorial on writing extensions. In fact, all UI tools you see on the bottom toolbar are extensions that can be easily used as reference to implement your own tools.

An extension can be loaded by using the viewer method `loadExtension`:

```
<script src="./templateExt.js"></script>

...

<script>

    // ...

    viewer.loadExtension('TemplateExtension');


    // ...
</script>
```

**Tip**: *Although extensions are generally supporting all versions for the viewer, sometimes it is important to know the specific version. As a debugging tool, you can use the **LMV_VIEWER_VERSION** to determine the version of the viewer you are using at runtime.*

## Loading a “system” Extension – Example WebVR

Let’s start by showing how to load a system extension. This example uses the WebVR extension, which is not listed in the officially supported extensions, but has been available for more than a year now.

```
var options = {
    'document' : modelName,
        'env':'Local',
    };

var viewerElement = document.getElementById('viewer');
var viewer = new Autodesk.Viewing.Private.GuiViewer3D(viewerElement, {});

Autodesk.Viewing.Initializer(options,function() {
    viewer.initialize();
    viewer.load(options.document);
    viewer.setLightPreset(lightPreset);
    viewer.loadExtension('Autodesk.Viewing.WebVR', Autodesk.Viewing.createViewerConfig());

});
```

**Tip:** *The [http://lmv.ninja.autodesk.com/](http://lmv.ninja.autodesk.com/) site is a testing site for the Forge Viewer behavior. Here you can test different versions of the viewer, and after loading a model, a section under the viewer will display the available “system” extensions and allow you to load/unload them on demand.*

## What is an Extension?

An extension is a new class container for code to provide specialized functionality to the Viewer. You will extend the Autodesk.Viewing.Extension class and override certain methods that will provide the hooks for the viewer to interact with your extension code. You will then need to register the extension with the system. Here is a simple example for the minimum requirements:

```
class TemplateExtension extends Autodesk.Viewing.Extension {
    constructor(viewer, options) {
        super(viewer, options);
        this.viewer = viewer;
    }

    load() {
        console.log('TemplateExtension is loaded!');
        return true;
    }
    unload() {
        console.log('TemplateExtension is now unloaded!');
        return true;
    }
}

Autodesk.Viewing.theExtensionManager.registerExtension('TemplateExtension',
    TemplateExtension);
```

This gives you the fundamental requirements for an extension.

**Tip:** *During development and testing to do more with the viewer, it is helpful to have a viewer instance at runtime. There is a global variable called:* ***NOP_VIEWER****. At any time during the runtime of a viewer instance, you can get this global and perform viewer tests at runtime.*

## What can you do?

We are now going to focus on using the three.js library to enhance the visual aspects of the viewer. The functionality we will show examples of, includes:

- Create custom geometry
- Custom Materials
- Transform SVF model geometry
- Create Animation

## Adding custom geometry and custom materials

The process of adding a model to Forge Viewer and simple customization of the experience is very well covered in the Viewer documentation. For more complex customizations, like adding custom geometry, materials, shaders, transformations etc. some knowledge of three.js is required.

In the case of a 3d visualization, there is an analogy with making a film, where in order to show something you would need the following things:

- ***Scene***– a container of all things we need for the performance;
- ***Camera*** – usually of type Perspective or Orthographic, that we will use to see the world/performance;
- ***Actors*** – the components/geometry that will be positioned and animated through a series of transformations (move/rotate/scale).
- ***Lights*** – that will bring “into light” the actors from the ominous dark;

For our immediate needs, we will concentrate our attention on two components from the above list: ***Scene*** and ***Actors***. Everything is already taken care by Forge Viewer, but knowledge of these two components is what we need for more advanced customization.

Apart from many internal custom parts, a Forge Viewer instance already contains a scene that is essentially a three.js scene, but it also contains a lot of additional features necessary for optimized visualization of large models (containing thousands of components) and consequently some of those parts cannot be accessed directly. Some access, however, is indirectly available through interfaces (i.e. node access).

Many things that we would do in a “vanilla” three.js scene, we can also do in the Forge scene, and the first step would be to bring a new actor (in this case geometry as a mesh) into the scene: In the case of “vanilla” three.js, to add an actor to a scene, you would do:

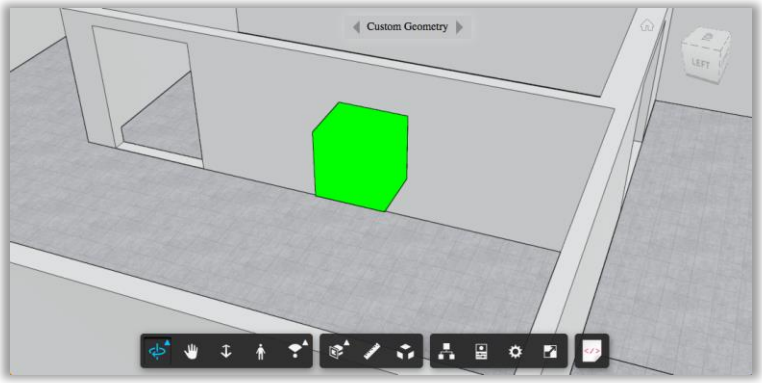
```
let geometry = new THREE.BoxGeometry( 1, 1, 1 );
let material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
let cube = new THREE.Mesh( geometry, material );
scene.add( cube );
```

In the case of Forge Viewer, the steps are similar:

```
let geometry = new THREE.BoxGeometry( 1, 1, 1 );
let material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
let cube = new THREE.Mesh( geometry, material );
viewer.impl.scene.add(cube);

viewer.impl.sceneUpdated(true);
```

The main difference is that the scene is buried deep inside the Viewer (viewer.impl.scene) and it also requires an additional step (viewer.impl.sceneUpdated) to inform the viewer that there is a new Actor to show in the scene.

Easy? Well it's a start… If we take a closer look at the newly added geometry, we may observe that it doesn't look right, and even if we give it a transparent material, it still looks the same:




To understand why it is not appearing as you would expect, pay attention to what it takes to define a new actor (in this case a cube) and we can see that it needs two things:

- ***a shape/form*** – the geometry of the new component;
- ***a costume*** – the material of the actor, or in other words how it reflects/refracts the light.

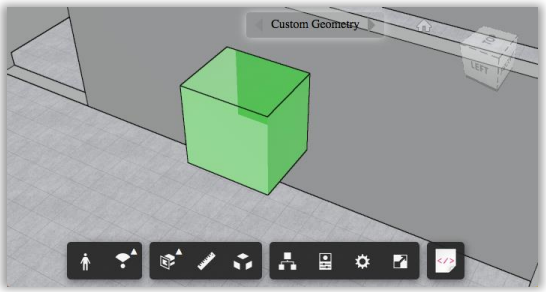
The idea of "how it reflects/refracts the light" is very important here as (along with the lights), a material defines how something looks.

We can create a shape, add a material and encompass this into a mesh to be added to the scene, but we didn't say anything about lights – a component necessary to be able to see the new actor. The Forge Viewer already contains the lights, but the problem is that the new actor brings its own costume/materials and the lights from the Forge Viewer are not aware of this new costume.

The solution is pretty simple, we just have notify the viewer of this new costume by registering the new material through Forge Viewer's material manager:

```
const my_material = new THREE.MeshPhongMaterial({
    specular: new THREE.Color(color),
    side: THREE.DoubleSide,
    color,
    transparent: true,
    opacity: 0.5
});

const materials = viewer.impl.getMaterials();

materials.addMaterial("my_super_material", my_material, true);
```

The stage is set, the lights are on, all actors are placed, the show goes on:




To summarize, in order to add some new three.js geometry to the Forge Viewer scene:

1. Create your geometry
2. Create the material
3. Construct a Mesh using the geometry and material
4. Add the object to the scene
5. Add the material to the Forge Viewer materials set
6. Update the Forge Viewer scene

Now, what about changing the material of already an existing component? This is not an easy task due to how Forge Viewer structures the meshes in the scene. Unlike the “vanilla” three.js meshes, where we can just say something like:

```
cube.material = new_material;
```

and assign a new material, in the Forge Viewer everything is structured in a way that favors performance. But, as mentioned before, some of it can be accessed and set through interfaces. If we break down the Viewer structure, it makes easy to abstract everything into sets of functions and to build-up a personal library.

To understand the process, we need the following named components from the Forge Viewer:

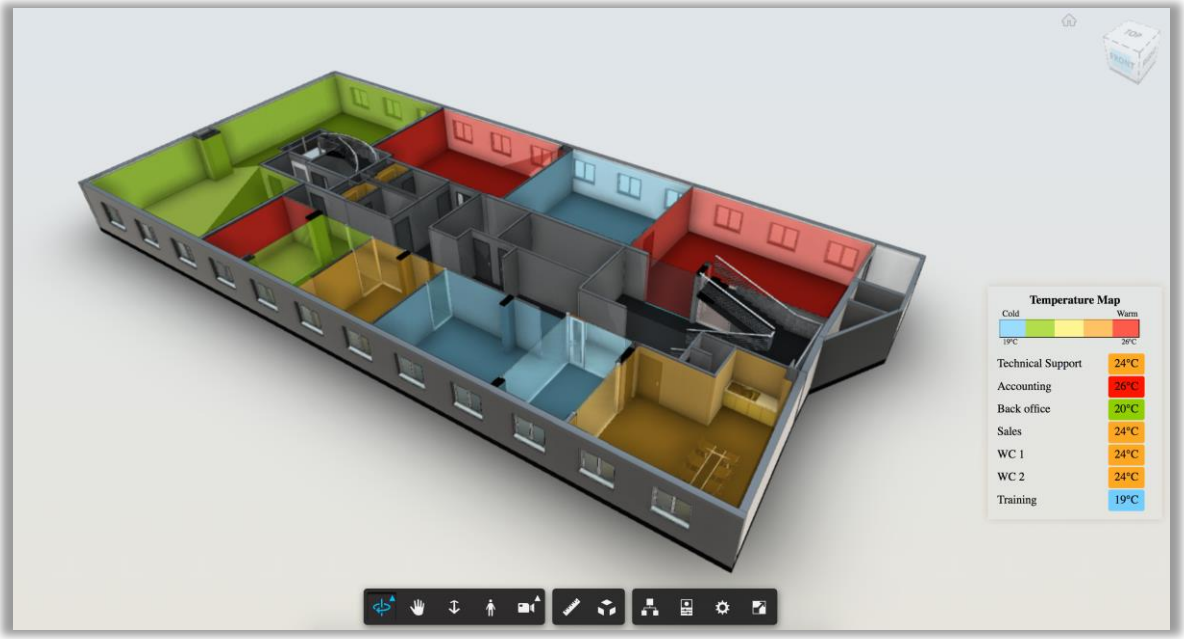
1. **nodeID** – any component in the scene has a unique id, which can be used for further references;
2. **instanceTree** – a container holding the structure and components of your model;
3. **fragment** – part of the geometry corresponding to a specific node that holds information on where that part is located and how it looks;

Then to change the material on a Forge Viewer geometry component:

1. Get the nodeID of the interested component. For example, you can do that by selecting the needed component and using: ***viewer.getSelection();***
2. Get the instanceTree, accessing the model data: ***viewer.model.getData().instanceTree***
3. Setup the new material (and register it as mentioned above), then assign it to all fragments associated to the desired nodeID.

```
const nodeID = 412;
const color = 0x73CEFF;

// create material
const material = new THREE.MeshPhongMaterial({
    side: THREE.DoubleSide, reflectivity: 0.0,
    flatShading: true, transparent: true,
    opacity: 0.5, color
});

const materials = viewer.impl.matman();

// register material
materials.addMaterial("MyCustomMaterial", material, true);

// get all fragments and asign new material to them
tree.enumNodeFragments(nodeID, (fragId) => {
    viewer.model.getFragmentList().setMaterial(fragId, myMaterial);
});

// inform the viewer that there were changes to the scene
viewer.impl.invalidate(true);
```

This allows you to change the material of any component and you can go beyond that of just changing the color or texture statically to setup a scene. What if you wanted to change it at runtime? A UI control on the component's color and/or transparency is enough to bring an interactive and visual override to the default Viewer behavior to make your model and viewing experience even more useful to the customer. For example, controlling the room color in the model based on temperature sensor in each room could be a very nice experience:

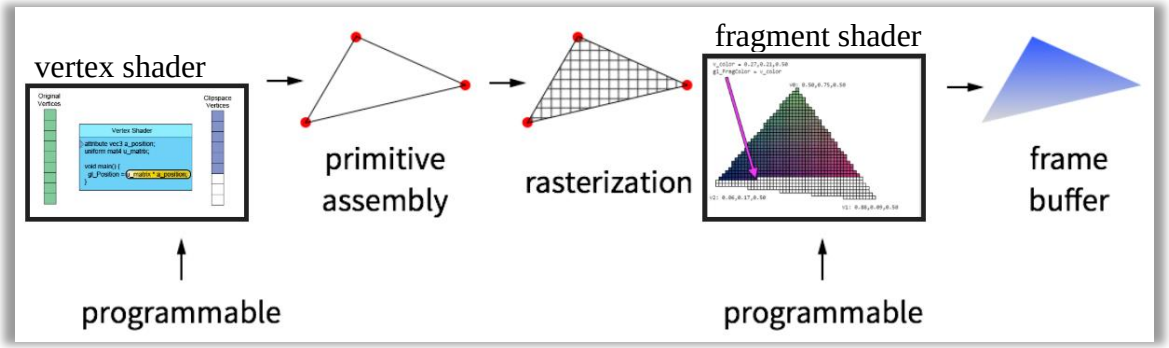



**Tip:** *Notice that we used **viewer.impl.sceneUpdated** method in the first example, and **viewer.impl.invalidate** in the second. Although this is not well documented, there is a subtle difference. When using invalidate, the Viewer will reprocess the entire scene and can cost performance, so use it wisely. The sceneUpdated will only update things that were changed (or added in our example). In case of materials it is better to use invalidate because the material list is outside the scene itself.*

## Power of shaders

Controlling the material color an opacity is just the tip of the iceberg. Three.js and Forge Viewer consequently allows you to go even deeper and control at GPU level how a vertex is projected and how each pixel is represented on the screen.

If we review the real-time graphics pipeline, where the geometry data is processed and the final picture appears:




We can notice that there are 2 steps where we can fine-tune the process, through writing microprograms on how the input should be processed at this step. These microprograms are called shaders and in three.js and Forge Viewer consequently, the way of creating a material that accepts shaders could not be simpler:

```
const material = new THREE.ShaderMaterial({
    vertexShader: myVertexShader,
    fragmentShader: myFragmentShader,
});
```

The first step in the pipeline is the **vertex shader** and it's main responsibility is to specify how the vertices of your geometry should be projected on the canvas:

```
void main() {

    vec4 mvPosition = modelViewMatrix * vec4( position, 1.0 );
        gl_Position = projectionMatrix * mvPosition;

}
```

Unless you want to add some distortion (like skewing effect) to your components, the use of Model View Projection Matrix is more than sufficient to give volume to your components and for our purpose, this should be more than enough.

**Tip:** *Notice that we are using **modelViewMatrix** and **projectionMatrix** variables, but we never specified it. This is a little help from the three.js library, which along with some other built-in uniforms and attributes, comes free and are filled for you with all needed data about the camera field of view, camera position, frustum params and other goodies.*
*Check [https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram](https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram) for full list of built-ins.*

After all vertices were assembled and the geometry was rasterized (determined which pixel will be filled), the next programmable step is the pixel shader (also known as fragment shader) where we can define the rules on how each pixel will be colored.

The simples microprogram that can be at this step is to color every pixel in one color, as in this case an opaque blue:

```
void main() {

    gl_FragColor = vec4(0,0,1, 1);

}
```

This step is the one we can use to determine how a component looks and it becomes even more important when you want to add custom texture to a component, as you are able to control how this texture should look and change under certain conditions.

The only thing that is left is to pass the information on texture to this fragment shader, and here is where the **uniforms** comes into play:

```
const uniforms = {
    texture1: {
        type: "t",
        value: THREE.ImageUtils.loadTexture( "../img/plan.png" )
    }
};

const material = new THREE.ShaderMaterial({
    uniforms: uniforms,
    vertexShader: myVertexShader,
    fragmentShader: myFragmentShader
});
```

Uniforms is the way of passing data to those shaders and are called ***uniforms*** because they are written in the constant part of the memory and cannot be changed:

***Vertex shader***

```
varying vec2 vUv;

void main() {

    vUv = uv;

    vec4 mvPosition = modelViewMatrix * vec4( position, 1.0 );
        gl_Position = projectionMatrix * mvPosition;
}
```

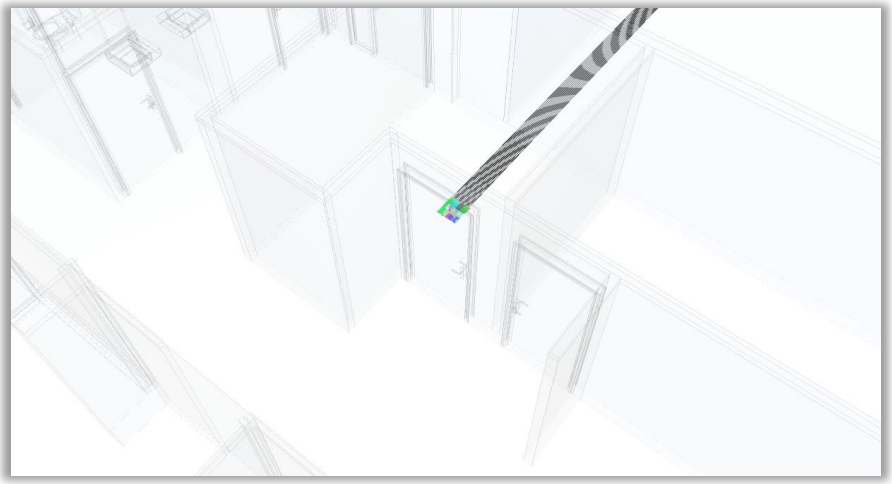
***Fragment shader***

```
uniform sampler2D texture1;
varying vec2 vUv;

    void main() {

    gl_FragColor = texture2D(texture1, vUv);
}
```

Also, there is no problem of passing the data between shaders and this is where the ***varying*** keyword comes into play, signaling that this is a variable chunk of the memory and it can be freely modified, and use to transmit data between shaders. As in the above shaders, the vertex shader transmits through vUv variable, the position of the vertex relative to the given texture and the fragment shader will interpolate the color for each pixel based on position of each vertex (relative to the texture). The only thing that these shaders need is the uv position, which might be source of the problem.

Imagine that in our scene we create a box through the above mentioned method, create the ShaderMaterial with abovementioned shaders and pass it a texture, aiming to create a sort of info panel with building plan:

It should work like a charm, but how it knows how to stretch the image? It turns out that upon box creation, three.js specified also the **uv data** for that object that could be simplified with words like “upon receiving the texture, stretch it from corner to corner”.
However, if we try to apply the same materials with same shaders to one of our components (g.e the floor component), then we can see that it has no idea how to position the texture on this object:



This is exactly the case where fine-tunning at the shader level is the ideal solution.
To fix the problem, it would be nice to pass to the shaders the width and the height of our component. This can be achieved from the bounding box info:

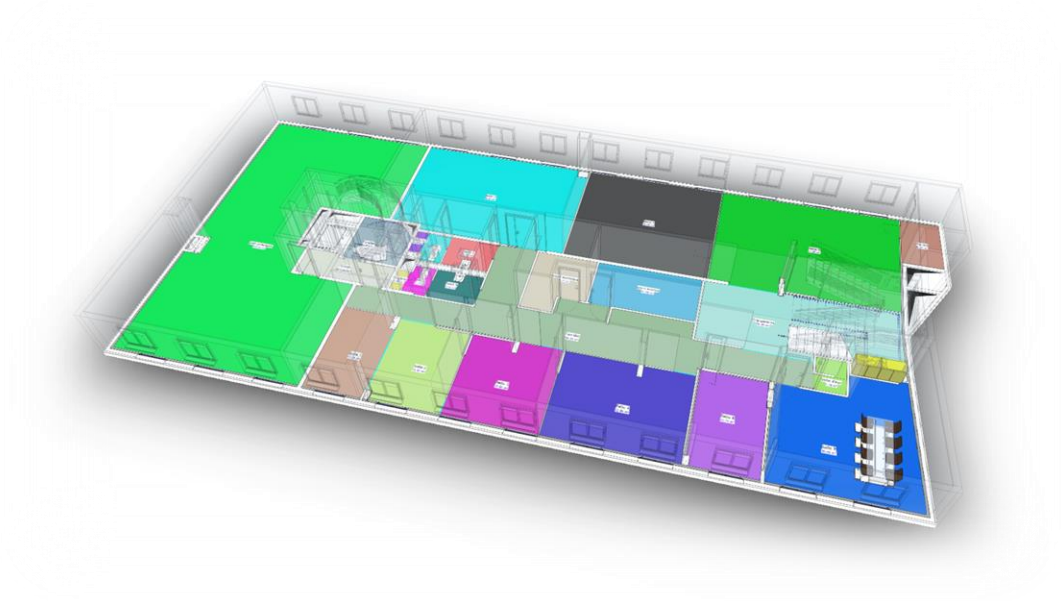
```
let floorBox = new THREE.Box3();
this.viewer.model.getFragmentList()
    .getWorldBounds(floorFragmentID, floorBox);

const width = Math.abs(floorBox.max.x - floorBox.min.x);
const height = Math.abs(floorBox.max.y - floorBox.min.y);

let bounds =  {
    width: width,
    height: height};
```

The width and height are needed, to be able to compute the relative position of each vertex uv to the given texture, and here is exactly where the uniforms come handy again:

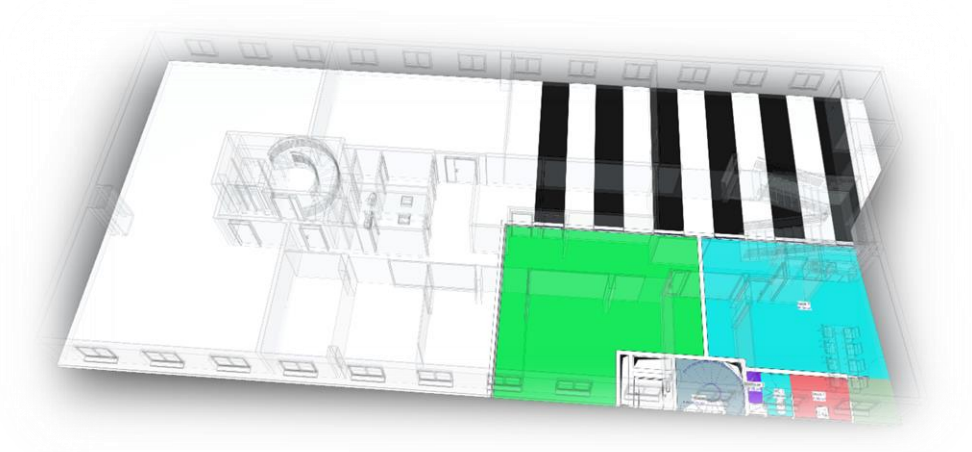
```
// vertextShader
varying vec2 vUv;
uniform float width;
uniform float height;

void main() {

    float OFFSET_X = 0.5;
    float OFFSET_Y = 0.5;

    vec3 projection = vec3(position.x, position.y, 0.);

    vUv = vec2((projection.x) / width + OFFSET_X,
        (height + projection.y) / height - OFFSET_Y);

    vec4 mvPosition = modelViewMatrix * vec4( position, 1.0 );
        gl_Position = projectionMatrix * mvPosition;
}


// fragment shader
uniform sampler2D texture1;
varying vec2 vUv;

        void main() {
        gl_FragColor = texture2D(texture1, vUv);
}
```

```
const uniforms = {
    width: {
        type: 'f',
        value: bounds.width
    },
    height: {
        type: 'f',
        value: bounds.height
    },
    texture1: {
        type: "t",
        value: THREE.ImageUtils.loadTexture( "../img/plan.png" )
    }
};

const material = new THREE.ShaderMaterial({
    uniforms: uniforms,
    vertexShader: myVertexShader,
    fragmentShader: myFragmentShader,
    side: THREE.DoubleSide,
});
```

This might look overwhelming at first, but once you give it a proper attention and understand the magic behind it, it becomes piece of cake and a very powerful tool for interesting ideas:

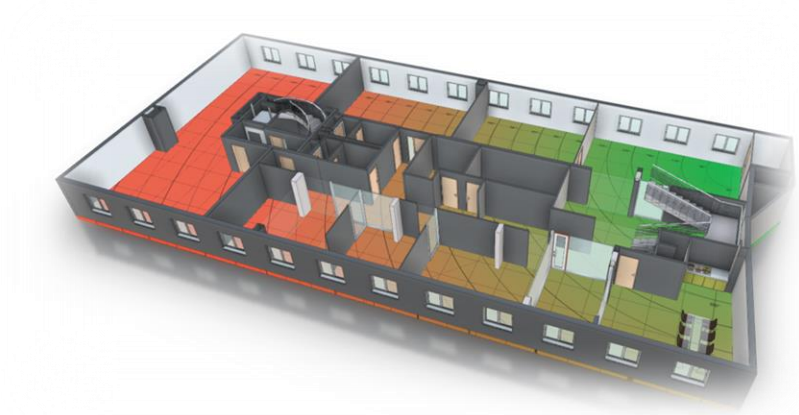


The best way of learning the shaders is to experiment with it and for sure the above presented shaders raised a lot of questions like “what this thing with OFFSET_X is?”. The best way to find out and understand it is to play with numbers and see why those constants are needed:

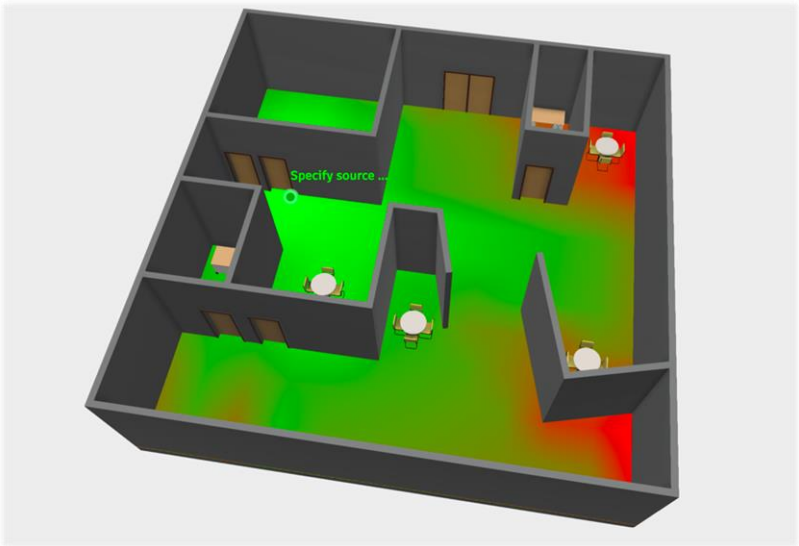


Which will lead you to the next step, where these constants could not be constants, but rather uniforms received externally from different events like mouse clicks.

In this case, with some specially designed texture, mixed with some local data, you can add some interaction to your model and upon click on a component (g.e. floor component), the position of the texture can change:



At this step, this idea could be pushed even further by adding some logic for signal attenuation while getting into consideration the positions of the walls and the materials they are made of. In this case the shaders becomes indispensable and I hope you start to sense their power.



## Adding custom controls and animations

The Forge Viewer is optimized to progressively load and present models with high number of components (order of thousands). This explains the somewhat cumbersome internal structure of model and why we cannot just have a node and move it all around as we would do with “vanilla” three.js mesh object.

To understand the process, we need nearly the same named components from the Forge Viewer, as in case of changing the material:

1. **nodeID** – any component in the scene has a unique id, which can be used for further references;
2. **instanceTree** – a container holding the structure and components of your model;
3. **fragmentProxy** – a proxy to the geometry corresponding to a specific node that holds information on where that part is located.

If you want to get the current world position/rotation/scale of a component, or assign it a new position/rotation/scale (transformation), you will need access to the fragmentProxy for each fragment associated to the needed node, and this is done by enumerating the fragments, but in this case we will also get the fragment proxy of each material:

```
const nodeID = 412;

// get all fragments and get the transformation matrix
tree.enumNodeFragments(nodeID, (fragId) => {

    let fragProxy = viewer.impl.getFragmentProxy(viewer.model, fragId);

    let matrix = new THREE.Matrix4();

    fragProxy.getWorldMatrix(matrix);

});
```

The transformation matrix that is retrieved contains all the important information. For instance, to get the component’s world position it is enough to call ***matrix.getPosition()***, that will return you a vector with x, y and z. You can also decompose the matrix directly into transformation info:

```
var position = new THREE.Vector3();
var rotation = new THREE.Quaternion();
var scale = new THREE.Vector3();
matrix.decompose(position, rotation, scale);
```

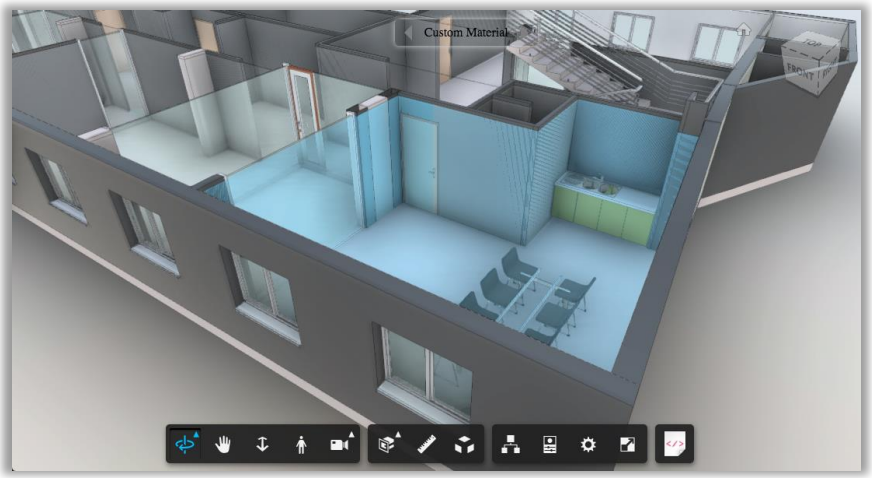
One thing that should be noted is that we are not working directly with geometry, but rather using the **fragment proxy** to get and even to update the information on the geometry.

If you check the fragmentProxy object you will notice that it holds a reference to position, quaternion (rotation) and scale, but these references are undefined, unless we ask to populate them and also serves as channels through which we can pass new values to the geometry:

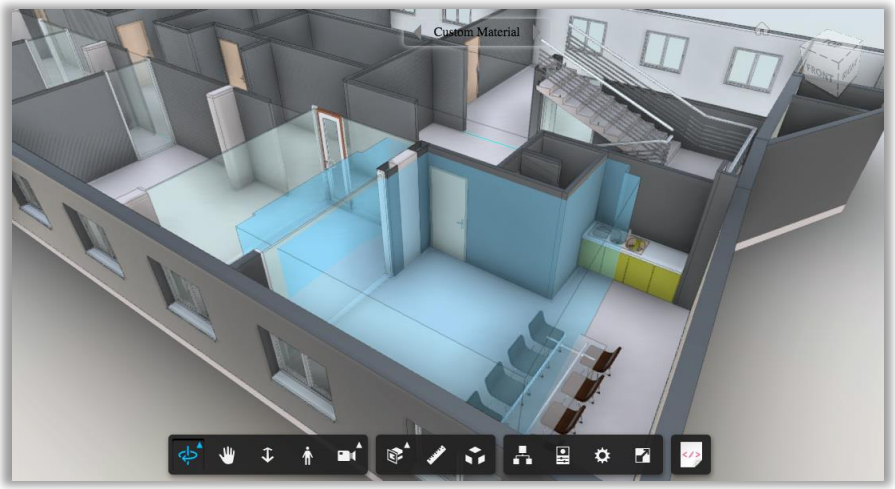
```
const nodeID = 412;

// get all fragments
tree.enumNodeFragments(nodeID, (fragId) => {

    let fragProxy = viewer.impl.getFragmentProxy(viewer.model, fragId);

    // ask to populate the transformation values of a fragment proxy
    fragProxy.getAnimTransform();

    // access directly the information
    console.log(fragProxy.position);
    console.log(fragProxy.quaternion);
    console.log(fragProxy.scale);

    // assign new transforms
    fragProxy.position = some_new_position;     // type THREE.Vector3
    fragProxy.quaternion = some_new_rotation;   // type THREE.Quaternion
    fragProxy.quaternion = some_new_scale;      // type THREE.Vector3

    // ask for new transforms to be applied
    fragProxy.updateAnimTransform();

    //inform Forge Viewer that the scene was updated
    viewer.impl.sceneUpdated(true);
});
```

As mentioned before, the best approach here is to abstract all this complexity into a set of functions like ***getTransformation*** and ***assignTransformation***, adding to your library of functions.

The main complexity comes from the fact that we are dealing with global transforms and we should always keep this in mind. For example, if we want to scale down a geometry, like in this case a colored room:




We would expect that if we apply the above mentioned approach and assign to fragment's proxy scale parameter a new ***THREE.Vector3(0.9,0.9,0.9)***, in other words rescale it by 0.9 factor, then it will just uniformly scale in place, but no, we can notice a displacement:

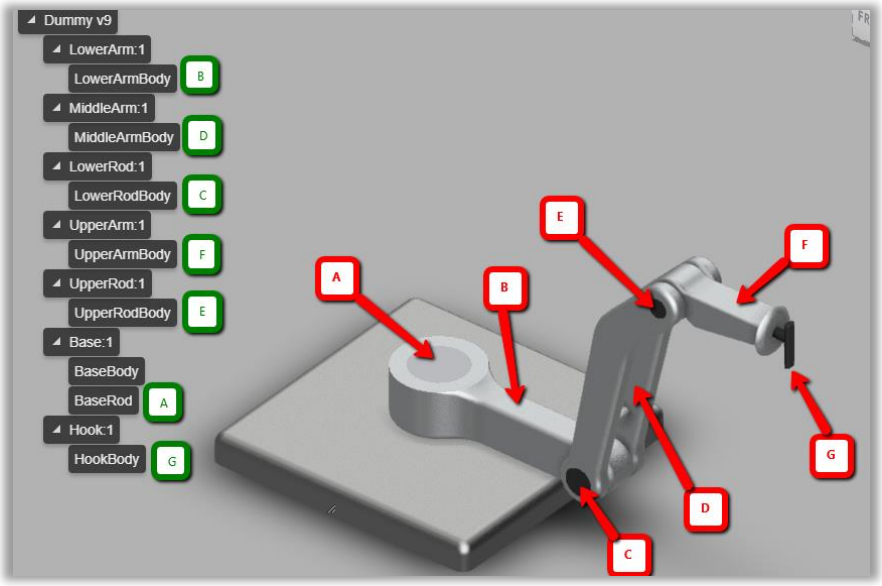



This is explained by the fact that the scale pivot point is the world's center instead of object's center and the usual approach for fixing this is to move the object to the origin, scale it and then move it back, which results in a very nice matrix chain and a lot of fun with matrix multiplications.

This comes even more interesting if you want to rotate a component around another component or create complex hierarchical transformations (i.e. forward kinematics), then for sure the approach of moving objects around, rotating them based on other component's position will make your work a bit unpleasant.

Fortunately, we can turn again to three.js for help and make this problem easier.

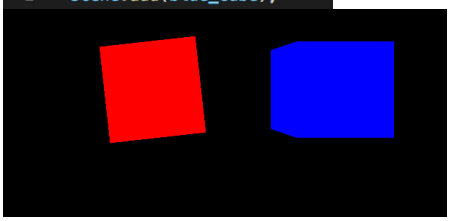
Let us assume that we have the model of a robotic arm:




Where we want to implement rotation of component B (LowerArmBody) around the component A (BaseRod).

To understand how to solve this problem, let us look at a simple three.js scene where we have a red cube (our pivot) and blue cube (our arm), both being added directly to the scene:

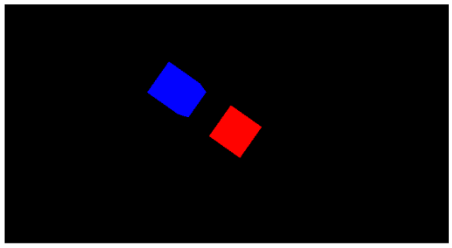
```
scene.add(red_cube);
scene.add(blue_cube);
```



If we rotate the red cube, it rotates normally, but it doesn't affect the blue cube, because the parent of the blue cube is the scene itself.

However, if instead of adding the blue cube directly to the scene, we add it to the red cube, we create a parent-child relation and any rotation done to the red cube will affect the blue cube:

```
scene.add(red_cube);
red_cube.add(blue_cube);
```



From blue's cube perspective, red cube now is the center of the universe.
It would be very nice if could do the same with model's components in Forge Viewer, but we can't because again the Forge Viewer is optimized to progressively load and present models with high number of components (order of thousands) and for this kind of work it needs its own way of storing and sorting of the geometry information.

As we already saw with adding new geometry to the Forge Viewer, there is no problem of mixing three.js functionality with Forge Viewer and we have all necessary ingredients for the component rotation algorithm:

1. add red cube to the scene and place it where the rotation pivot should be (in our case the center of the base rod):

```
let pivotBaseID = 15;
let dummy_center = new THREE.Mesh(new THREE.BoxGeometry(5, 5, 5),
                                  new THREE.MeshBasicMaterial({ color: 0xff0000 }));
let center_position = this.getFragmentWorldMatrixByNodeId(pivotBaseID).matrix[0].getPosition().clone();
dummy_center.position.x = center_position.x;
dummy_center.position.y = center_position.y;
dummy_center.position.z = center_position.z;
this.viewer.impl.scene.add(dummy_center);
```

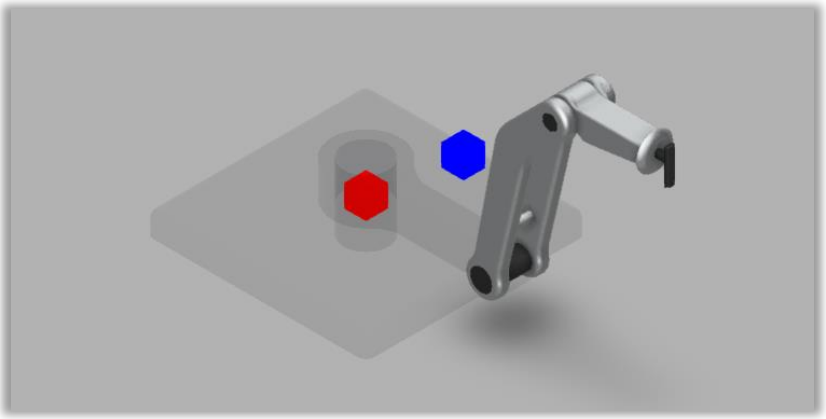
2. add blue cube and place it at position of "need to move/rotate" component (in our case LowerArmBody), but this position should be relative to rotation pivot:

```
let mainArmID = 4;
let dummy_mainArm = new THREE.Mesh(new THREE.BoxGeometry(5, 5, 5),
                                   new THREE.MeshBasicMaterial({ color: 0x0000ff }));
let element_position = this.getFragmentWorldMatrixByNodeId(mainArmID).matrix[0].getPosition().clone();
dummy_mainArm.position.x = -element_position.x + Math.abs(element_position.x - center_position.x);
dummy_mainArm.position.y = -element_position.y + Math.abs(element_position.y - center_position.y);
dummy_mainArm.position.z = -element_position.z + Math.abs(element_position.z - center_position.z);
dummy_center.add(dummy_mainArm);
```

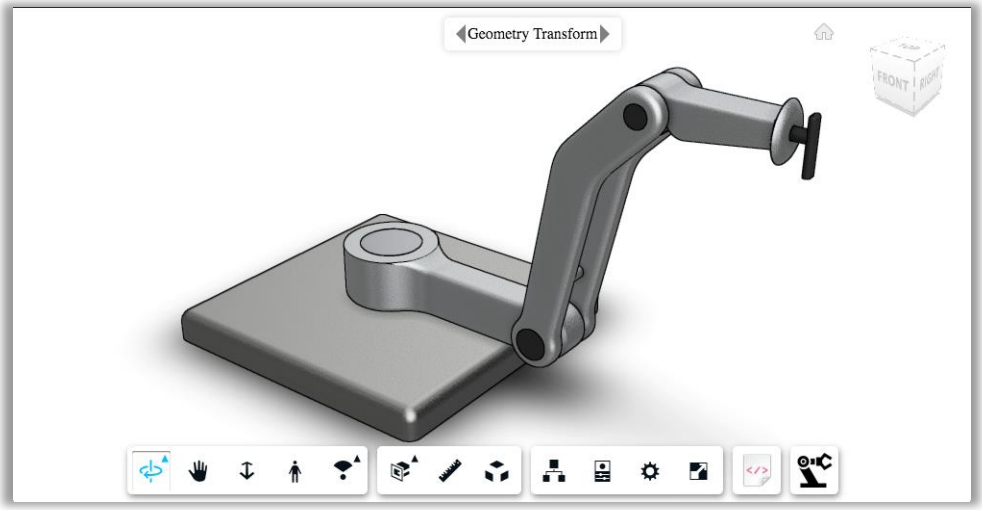
3. on each rotation of red cube, read the world matrix for the blue cube and assign the transform data to the needed component through fragmentProxy:

```
assignTransformations(dummy_mainArm, mainArmID);
function assignTransformations(ref_obj, nodeId) {
    ref_obj.parent.updateMatrixWorld();
    let position = new THREE.Vector3();
    let rotation = new THREE.Quaternion();
    let scale = new THREE.Vector3();
    ref_obj.matrixWorld.decompose(position, rotation, scale);

    tree.enumNodeFragments(nodeId, function (frag) {
        var fragProxy = viewer.impl.getFragmentProxy(viewer.model, frag);
        fragProxy.getAnimTransform();
        fragProxy.position = position;
        fragProxy.quaternion = rotation;
        fragProxy.updateAnimTransform();
    });
};
```

Thus, anytime we rotate the red cube now, we read the world position of the blue cube and assign it to the appropriate component:



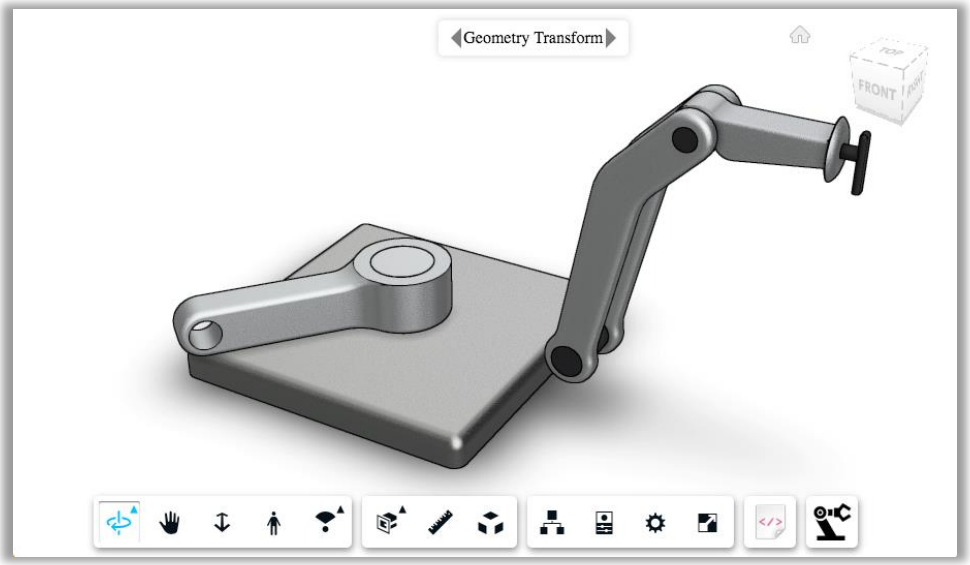
To understand how much time we saved with this approach, it helps to look at the data of a fragment proxy before and after rotation:




```
fragId: 2
▶ frags: d {model: d, fragments: {…}, geoms: d, pagingProxy:
▶ position: c.Vector3 {x: 0, y: 0, z: 0}
▶ quaternion: c.Quaternion {_x: 0, _y: 0, _z: 0, _w: 1}
▶ scale: c.Vector3 {x: 1, y: 1, z: 1}
```

In this state, the position, rotation and scale is at rest.

However, if we rotate 230 deg (or -130 deg):




```
fragId: 2
▶ frags: d {model: d, fragments: {…}, geoms: d, pagingProxy: undefined, isFixedSize: true,
▶ position: c.Vector3 {x: -36.153358459472656, y: 0, z: 16.85858726501465}
▶ quaternion: c.Quaternion {_x: 0, _y: 0.90630779410926, _z: 0, _w: -0.4226182555553163}
▶ scale: c.Vector3 {x: 1, y: 1, z: 1}
```

the position and rotation data look quite scary and you don't really care how this was achieved as long as everything looks right. With this simple technique you have ability for more complex transformations, like implementing forward (and maybe even inverse) kinematics.

Now that you know "the secret sauce", all these techniques might not look advance to you anymore, yet, be it added custom geometry, custom materials or simple transformations – all this individually or in combination can add value to your model and viewing experience.

## Sample Code

Code: https://github.com/apprentice3d/SD226781-Samples
Live: https://apprentice3d.github.io/SD226781-Samples/

## Autodesk Forge Resources

***Web:*** https://forge.autodesk.com

***Learning Resources*:**
Getting Started: https://forge.autodesk.com/developer/getting-started
LearnForge Tutorials: https://forge.autodesk.com/LearnForge
GitHub: https://forge.autodesk.com/GitHub
AR/VR Toolkit: https://forge.autodesk.com/ARVRToolkit
Stack Overflow: https://forge.autodesk.com/Stack

***Accelerators*:**
https://forge.autodesk.com/accelerator

***Community:***
- Twitter: @AutodeskForge
- Facebook @AdskForge
- YouTube: https://www.youtube.com/playlist?list=PL_6ApchKwjN9CZCqUl4RZrsyDvnTV1Jgb
- Blog: https://forge.autodesk.com/blog